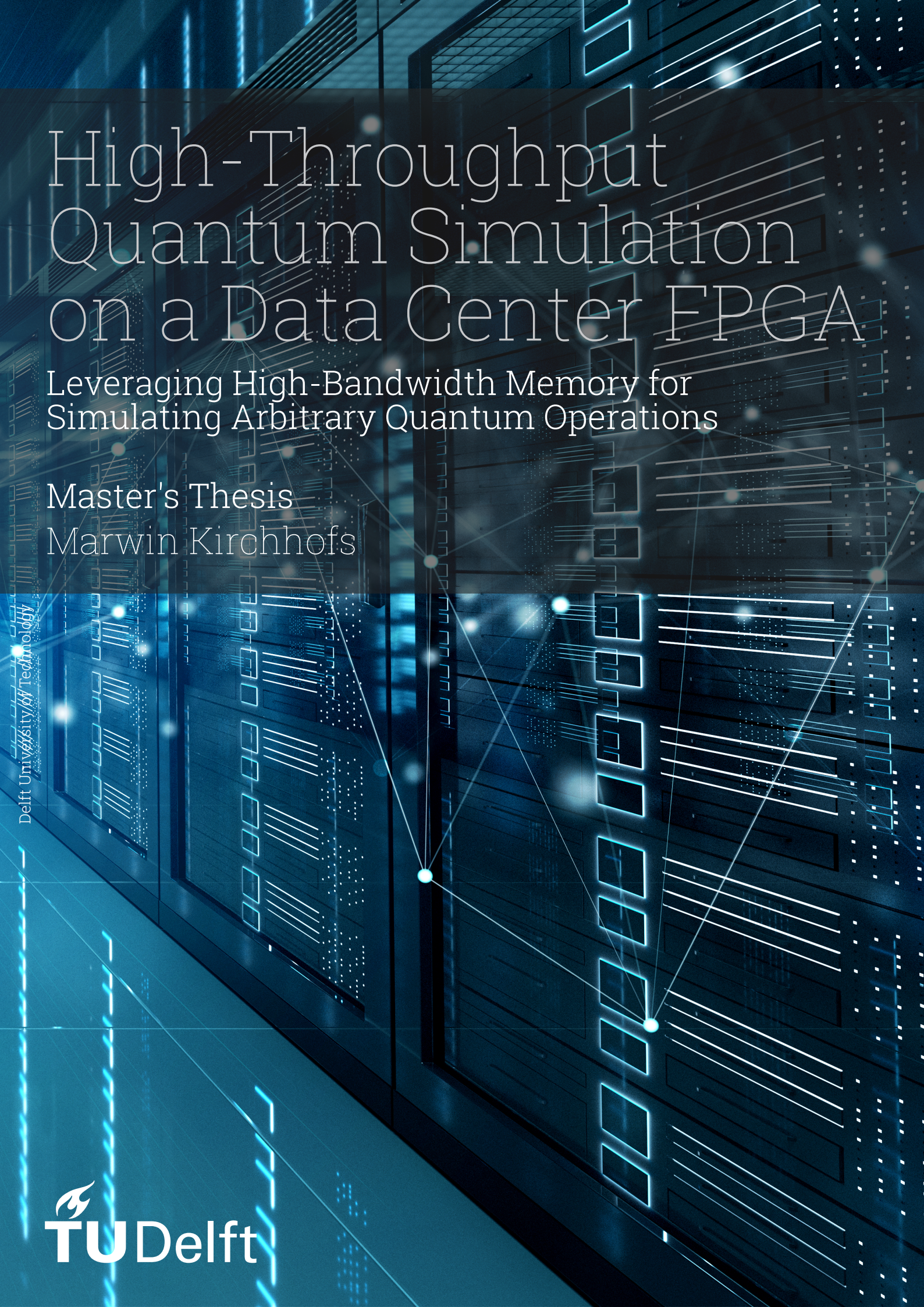


High-Throughput Quantum Simulation on a Data Center FPGA

Leveraging High-Bandwidth Memory for Simulating Arbitrary Quantum Operations

Master's Thesis

Marwin Kirchhofs

Delft University of Technology

TU Delft

High-Throughput Quantum Simulation on a Data Center FPGA

Leveraging High-Bandwidth Memory for Simulating Arbitrary Quantum Operations

by

Marwin Kirchhofs

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday February 13, 2026 at 09:30 AM.

Student number:	5616166	
Project duration:	April 1, 2024 – February 13, 2026	
Thesis committee:	Prof. Stephan Wong	TU Delft, Chair
	Prof. Sebastian Feld	TU Delft, Core Member 2
	Prof. Georgi Gaydadjiev	TU Delft, Core Member 3

Cover: Connection Network in Dark servers datacenter ruimte opslagsystemen 3D rendering by sdecoret under iStock standard license (Modified)
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Summary

Quantum Simulation is a crucial tool on the path towards real-world Quantum Information applications at scale - both for Quantum Computing, and for Quantum Networks. A substantial challenge in Quantum Simulation lies in the inherent exponential scaling of computing complexity and required memory resources, due to the nature of Quantum Information. This thesis demonstrates how DDR-based, parallel computing accelerators can be leveraged for achieving a significant speedup over conventional systems on simulating quantum state vectors - maximizing data throughput, while allowing arbitrary operations at any time (including quantum network simulation). As the algebraic foundation of this work, a novel scheme for state vector representation and computation is derived, which combines the parallelizability of conventional matrix multiplication with the efficiency of sparse matrix multiplication, eliminates the memory footprint of large gate matrices, and relies on purely sequential memory accesses. Afterwards, the novel scheme is implemented as a scaled-down prototype on an HBM-equipped Data Center FPGA accelerator platform, which in experiments already outperforms a high-performance CPU node on many scenarios. Lastly, the thesis describes in detail the necessary steps for transitioning from the prototype to fully exploiting the chip's potential performance, and highlights how the FPGA's flexibility enables further usecase-specific functionality additions.

Contents

1 Introduction

Motivation

Computation based on leveraging Quantum Information promises to disrupt how digital information is processed in the foreseeable future, but poses significant challenges for today's research, prior to its maturity. The technology allows for exponential improvements in the runtime and memory complexity of impactful algorithms, while also setting new standards in information transmission security. However, there is a wide gap between currently available hardware, and ongoing application development. General-purpose Quantum Computing and Quantum Networks, at real-world application scale, have not been achieved yet. Meanwhile, considerable effort is already being spent on associated algorithms and electrical components. Researchers commonly employ software simulators to validate and quantify novel developments. These simulators however struggle with the inherent exponential nature of computation and memory demand in quantum simulations.

Looking into other resource-intensive and data-intensive computing applications indicates a path towards more capable quantum simulators. Numerous computing tasks experience significant limitations on a conventional computing system, but greatly benefit from hardware acceleration - with the most prominent example arguably being Machine Learning (ML) acceleration, due to the recent ubiquity of Artificial Intelligence (AI) systems. From a computational perspective, ML inference and quantum state vector simulation share noticeable similarities. First, they both at the core rely on matrix-vector or matrix-matrix multiplications, which is known to be highly parallelizable - although ML leans towards dense matrices, while quantum operations are frequently represented by sparse matrices. Secondly, they both require moving (potentially) large quantities of data between memory and compute units, which often only fit into DDR-type memory. Therefore, a typical (Von Neumann) ML accelerator relies on a large, closely coupled, high-speed DDR, paired with a highly parallel computation structure, to match the offered memory bandwidth. The computational similarities motivate the idea of using that same concept for powering hardware-accelerated simulation of quantum information applications.

Problem

Several designs have already been published specifically for quantum operation simulation, either in software, or in hardware. However, none of these manages to offer both computation efficiency - with respect to quantum state vector operations - and good scalability with on-chip Double Data Rate (DDR) bandwidth. Additionally, several hardware-based schemes are severely limited in maximum quantum state size, lack on-chip support for essential operations like measurements, or are not equipped to handle a system of multiple quantum states. Therefore, no existing design is appropriate for hardware-accelerated, large-scale, high accuracy simulation of arbitrary quantum operations.

This thesis studies the question arising from the highlighted inspiration and shortcomings: How can parallel, DDR-based computing accelerators be used efficiently to speed up arbitrary simulation of quantum information applications?

Thesis Overview

The goal of this work is to design and implement a prototype simulator, and reliably indicate its potential performance. That encompasses three elements. Firstly, a computation scheme for quantum state

vectors shall be devised, with full support for operations on multi-state quantum systems, that offers great parallelizability, and is optimized for high-throughput DDR operation. In the second phase, that scheme shall be implemented, validated, and evaluated in a prototype system, using a suitable acceleration platform and software front end. Lastly, a qualitative and quantitative study shall be conducted on the prototype's margin for improvement, with respect to various quantum application scenarios.

Performance is evaluated on individual quantum operations, considering both computation time, and result accuracy. A representative CPU-based High-Performance Computing (HPC) node serves as the baseline.

The thesis is structured as follows:

- Chapter 2 provides essential information on Field-Programmable Gate Arrays (FPGAs) accelerators and quantum computing simulation. It is outlined how FPGAs are utilized in the context of performing computation, with an emphasis on Xilinx FPGA accelerators. On the quantum computing side, the fundamental principles of qubits and quantum states are explained, and how these can be used for computation and network purposes. That concludes in the need for quantum simulators, and in a categorization of simulator designs. Lastly, the chapter gives an overview of the current status of FPGAs in Quantum Computing simulation.
- Chapter 3 defines a computation scheme that fulfills the above requirements, and is not tied to a certain type of hardware platform.
- Chapter 4 translates the algebraically described computation scheme into generic, high-level hardware building blocks. The focus lies on designing for custom logic circuits, which is motivated at the beginning of the chapter.
- Chapter 5 narrows the platform type down from a custom logic circuit to an FPGA. It converts the high-level hardware building blocks into an FPGA implementation, and extends the core design into a full quantum simulator back end, targeting an AMD/Xilinx Data Center accelerator card.
- Chapter 6 outlines the validation and profiling setup, and proceeds with presenting the in-hardware accelerator evaluation, in terms of accuracy and speedup.
- Chapter 7 discusses optimizations within the implemented prototype design, and gives an estimate on the expected achievable performance.
- Chapter 8 summarizes the findings along the thesis, to subsequently value them against the above formulated question.
- Chapter 9 elaborates on further development directions of the baseline prototype. In contrast to Chapter 7, the chapter focuses on functional extensions to the implemented prototype, instead of optimizations in the existing design.

2 Background

This chapter discusses relevant background information for the computing scheme and hardware design proposed in this thesis. The information depth is tailored towards an audience with a background in Computer Engineering or a related field. The chapter consists of two main topics, and concludes with giving an overview on related publications. Firstly, Sections 2.1 to 2.2 discuss specific FPGA components, technologies, and devices, that play a role during the thesis. General knowledge on FPGA functionality is assumed. Afterwards, Sections 2.3 to 2.4 are dedicated to quantum simulation. No preliminary knowledge with quantum information is assumed. Finally, Section 2.5 illustrates the current state of quantum simulation on FPGA, which lays the foundation for the thesis at hand.

2.1. FPGA Fundamentals

This section aims to provide the fundamentals on dedicated FPGA components required to follow the proposed design. However, the first section gives a relatively general, brief overview on architectural principles in FPGAs. The sections afterwards provide details specifically on memory cells (Section 2.1.2), and then on Digital Signal Processor (DSP) cells in the context of computation on FPGAs (Section 2.1.3).

2.1.1. Architecture

FPGAs internally are composed of a columnar structure of general-purpose programmable logic cells, flexible routing resources, and hard-IP cells for dedicated tasks, as depicted in Figure 2.1. The general-purpose cells are called Configurable Logic Blocks (CLBs). They internally consist of a number of Look-up Tables (LUTs) (arbitrary programmable n-input 1-output logic functions), paired with a register (see Figure 2.2). Generally relevant local hard-IP cells are DSPs, Block RAM (BRAM) or UltraRAM memory cells, and Clock-Management Tiles (CMTs). Next to these distributed cells, many FPGAs feature additional non-distributed hard-IP components, like Double Data Rate (DDR) memory or a processor subsystem. I/O cells and Gigabit Transceivers (GTs) (serial high-speed transceivers for communication via PCIe or raw serial lines, for instance) fall in neither category, because they are found along the periphery of the chip. Out of all named hard-IP components, the ones that are of particular importance in this thesis are DSPs, local Random-Access Memory (RAM), and DDR memory.

2.1.2. Memory

This paragraph gives an overview of FPGA memory types, categorized and ordered by density, latency, ports, and access time determinism (for more detailed information see [3]). First, Look-up Table Random-Access Memory (LUTRAM) (also referred to as “Distributed RAM”) leverages the FPGA’s LUTs to store small, distributed datasets, offering deterministic one-cycle write latency and same-cycle read latency. It only supports single-clock operation, up to $16\,\mathrm{bit}$ dataport widths - which decreases with the number of ports - and only one write port in any configuration. LUTRAM is ideal for fast, localized data storage and register-intensive applications. BRAM, and on several chips UltraRAM, form a second category, distinguished by the fact that these are dedicated memory cells, in contrast to LUTRAM. BRAM delivers deterministic latency of 1-2 clock cycles (depending on the configuration), and generally supports dual-clock dual read and write ports, which makes it suitable for medium-sized buffers, First-In

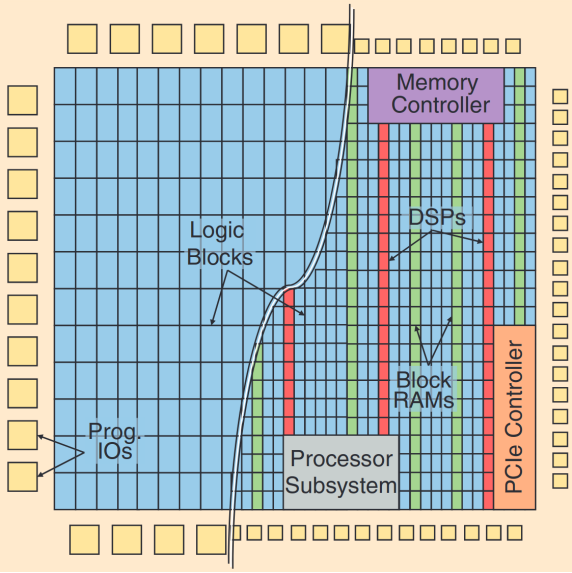


Figure 2.1: from [1]; FPGA architecture (initially and modern)

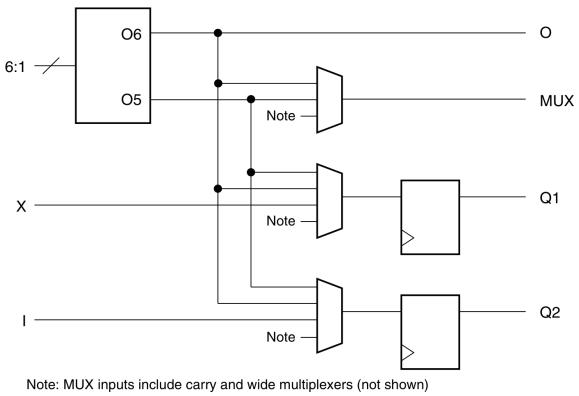


Figure 2.2: from [2]; CLB element with LUTs, Multiplexers, and Flip-Flops, in Xilinx UltraScale architecture (1 out of 8 elements in a CLB shown)

First-Outs (FIFOs), and data caching. For larger memory demands, UltraRAM offers significantly higher density, while still providing relatively local memory, and maintaining low-latency access (typically 2-3 clock cycles) - bridging the gap between BRAM and off-chip memory. UltraRAM is particularly advantageous for applications requiring substantial on-chip storage. In the remainder of the thesis, in the context of FPGAs, “local on-chip memory” will refer to BRAM and UltraRAM, as a collective term. Finally, DDR memory, interfaced externally to the FPGA, provides significantly higher storage capacity than the previous options, at the cost of higher access latency (tens to hundreds of clock cycles) and non-deterministic timing. In terms of DDR memory, FPGAs can feature either of Dynamic RAM (DRAM) or High-Bandwidth Memory (HBM) memory (explained in the next paragraph), or a combination thereof.

High-Bandwidth Memory (HBM)

HBM is a highly parallelized DDR memory, which can be found on high-performance FPGAs, and also Graphics Processing Units (GPUs). It is explained here in detail due to its significance in the proposed design. HBM consists of vertically stacked and interconnected DRAM dies. Each HBM stack is divided into channels, and each channel is further segmented into so-called Pseudo-Channels (PCs), as shown in Figure 2.3. Two PCs are operated by one Memory Controller (MC). In AMD/Xilinx FPGAs, interfacing with HBM can either be done via bare-metal interfaces (which necessitates essential DRAM controlling, like refreshing, in fabric logic), or via AXI buses. In both cases, the interface provides one read/write channel per PC, and therefore allows for highly parallelized operation. Additionally, HBM in AMD/Xilinx FPGA incorporates a bypassable hard-IP crossbar, in order to allow global addressing from any interface channel to any PC. Activating the crossbar comes at the cost of increased access latency, which comprises a fixed latency penalty, and an additional locality-depending portion.

2.1.3. DSP-based Computation

In FPGAs, DSPs are in many cases the predominant way to carry out algebraic computations, like matrix/vector multiplications or filter applications. They are hard-IP building blocks, with a certain level of parameterizability - either at design time, or at runtime. The centerpiece is typically a multiplier, since these are otherwise costly to implement in fabric logic. For supporting more data-intensive computations, it is commonly seen in FPGAs that a DSP and a BRAM column are placed closely together, with matching physical sizes of DSP and BRAM slices. This architecture allows for building efficient DSP computation structures with associated local BRAM data buffers.

Xilinx Ultrascale+ DSPs

The following paragraphs give a more detailed overview on operation and configuration options with the so-called DSP48E2 primitive. This is the hard-IP DSP unit found in high-end Xilinx Ultrascale/Ultrascale+

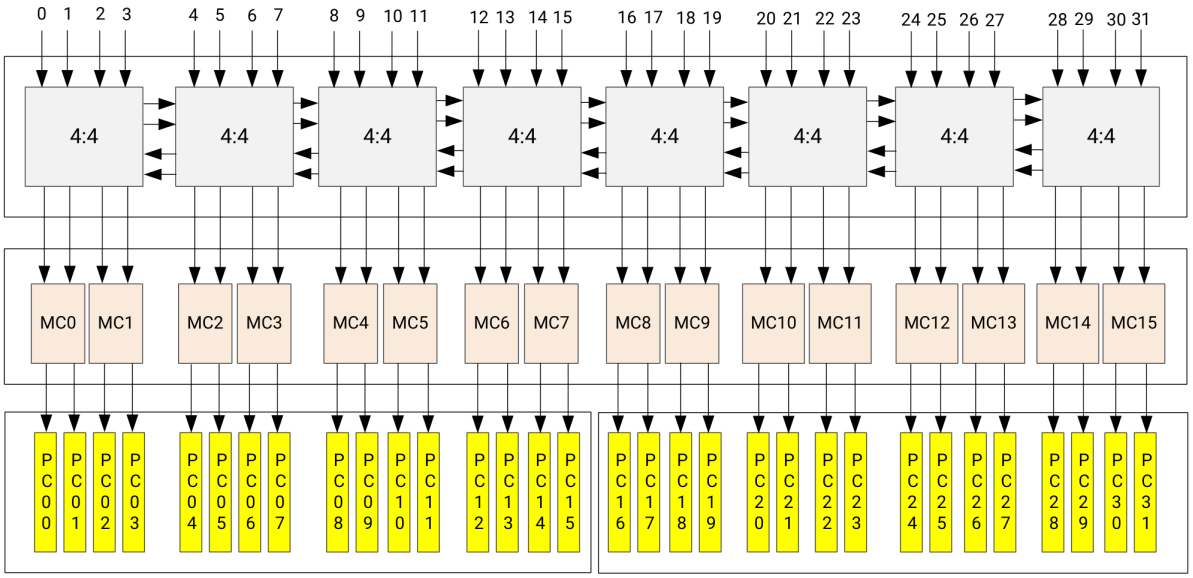


Figure 2.3: from [4]; architecture overview dual-stack HBM memory, as used in AMD Alveo FPGA platforms

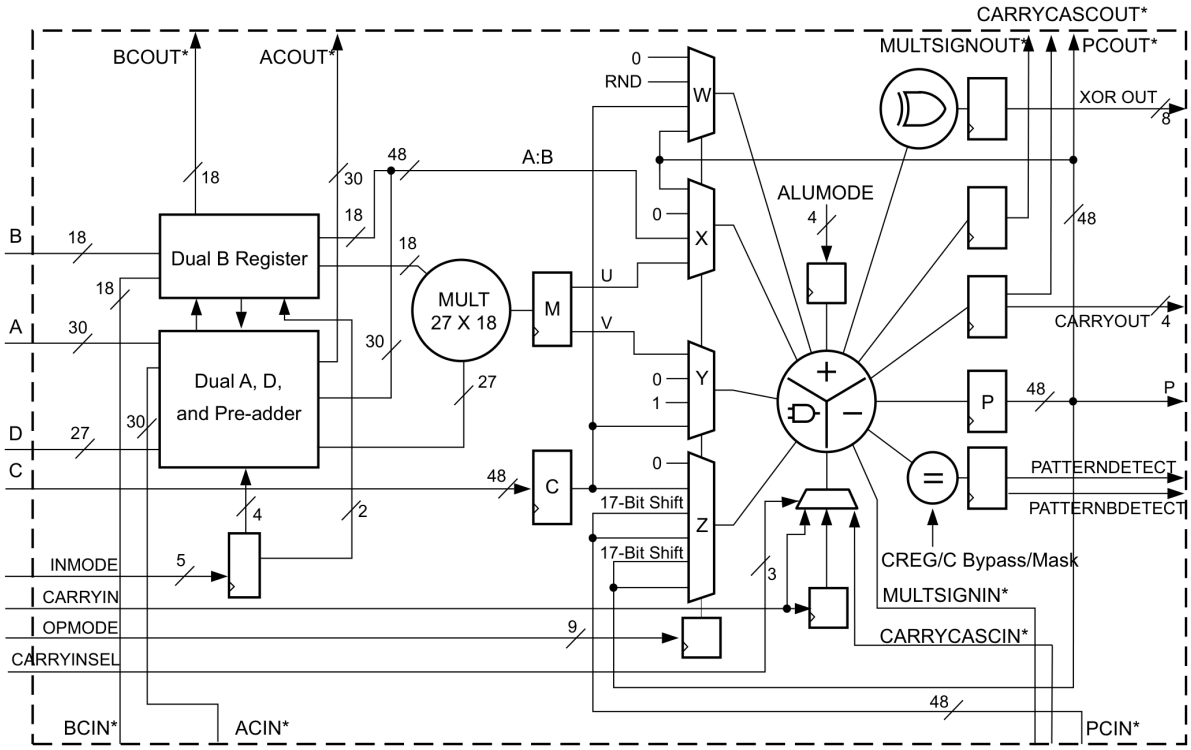


Figure 2.4: from [5]; internal circuitry of the DSP48E2 primitive in Xilinx (UltraScale) FPGAs

FPGAs, and the unit that is used exclusively in this thesis. Apart from the general architecture, only features are highlighted that are relevant to this thesis. A full discussion can be found in [5].

Figure 2.4 provides the circuitry of the DSP primitive, with configurable elements (multiplexers, register stages, operation control signals). In terms of general architecture, the DSP consists of two computation stages (two's complement integer multiplier and Arithmetical-Logical Unit (ALU)), and offers four input ports of different bitwidths. Pipelining registers can be activated or bypassed at multiple points between the stages. The multiplier itself is restricted in its operand bitwidths. The ALU allows for both arithmetical and bitwise operations, and can be fed up to four inputs, depending on the configuration of the W, X, Y and Z multiplexers. The second stage additionally features feedback paths from ALU output

to ALU input, for instance for result accumulation. In between of adjacent DSPs within a column, the chip features dedicated cascading traces for both the input and the output sections. Their predominant purpose is to facilitate efficient multiplier arrays, but they also allow for other applications like combining multiple DSPs into one unit, which effectively increases one of the two multiplier bitwidths by 17 or $18\,\mathrm{bit}$ per additional DSP (see [5], section “DSP48E2 Operation Modes”, pp. 46-49). The cascading traces are not accessible from general fabric logic.

In terms of configuration, several options have to be specified at design time (statically), while others can be controlled at runtime (dynamically). The most important static options are pipelining registers and cascading. The registers still offer dynamic control in the form of clock enable and reset signals, but they can not be dynamically activated or bypassed. Among the dynamic options, the most relevant characteristic is that both the datapath to the ALU, and the desired ALU operation, can be controlled dynamically (via the OPMODE and ALUMODE signals).

2.2. FPGA Accelerators

This section is dedicated to FPGA accelerator technology. After a brief overview, Section 2.2.1 specifically discusses any relevant aspect of AMD/Xilinx FPGA accelerator cards, since these devices are employed in the project presented in this thesis.

Acceleration on FPGAs

Next to their more traditional role as highly flexible circuits in digital control electronics, FPGAs can also thrive in the field of accelerated computing, with several hardware platforms developed specifically for that purpose. It is important to note however that in terms of raw Single-Instruction Multiple-Data (SIMD) computing power, FPGA accelerators are usually outperformed by GPUs - the most common dedicated hardware accelerator type. Nevertheless, specific applications allow for FPGA-based accelerators to compensate the disadvantage, by means of an implementation that is more specific to the algorithm at hand, and by allowing a more heterogeneous design (in terms of functionalities). Additionally, they are more suitable for multi-step data streaming applications, like package analysis and cryptographic tasks, which moreover benefit from the support for on-board communication I/O like ethernet ports. The design proposed in this thesis (from Chapter 4 on) demonstrates how the flexibility of an FPGA can be leveraged to accelerate a complex application in an adaptable way, such that it provides advantages to a range of scenarios that a GPU could not cover.

2.2.1. Xilinx FPGA Accelerator Cards

AMD/Xilinx is the largest manufacturer of FPGA accelerator hardware. Among the product portfolio, the Alveo accelerator devices are specifically designed for accelerating computing applications. The following paragraphs first discuss the system-level architecture of the Alveo cards framework, then the card and FPGA architecture, to finally target operating the card and designing for an Alveo system.

System Architecture

Figure 2.5 depicts the system-level architecture of an Alveo system. The cards are realized as PCIe modules, which are integrated into a host Central Processing Unit (CPU) system in the same way that a GPU typically is. A stack of drivers and interfaces in both hardware and software facilitates interaction between a “user-space” host application on the software side, and computing “kernels” in the FPGA on the hardware side. It is particularly important to note that “Global Memory” on the hardware side is managed by the platform drivers, and not part of the user’s kernel design. “Global memory” here mainly refers to off-chip DDR memory, but also on-chip memory can be designated as global memory, and handled by the platform. Considering off-chip DDR memory, all Alveo cards feature either $8\,\mathrm{GB}$ to $16\,\mathrm{GB}$ HBM (at least), or up to $64\,\mathrm{GB}$ DRAM, and sometimes both.

Card Architecture and Technology Node

The Alveo cards are implemented in Xilinx’s high-performance Ultrascale+ technology. This does not only determine the fabrication technology node, but also the respective Xilinx primitives library for on-chip hard-IPs like DSPs and BRAM. An import fact to be aware of is that Alveo cards are so-called Stacked Silicon Interconnect (SSI) devices (like other large AMD/Xilinx FPGAs). These are three-dimensional chips that consist of multiple vertically stacked dies, in order to enable a larger FPGA at still manageable

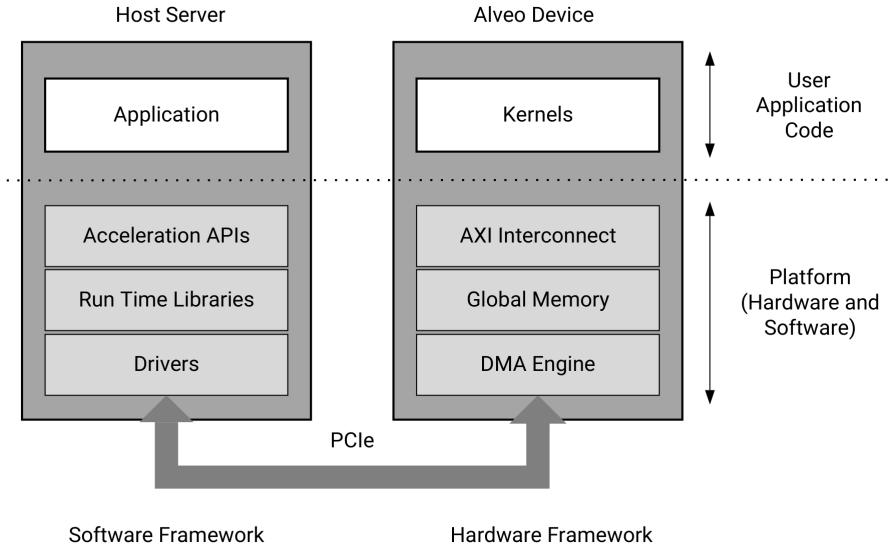


Figure 2.5: from [6]; Platform Overview of AMD Alveo Data Center accelerator cards

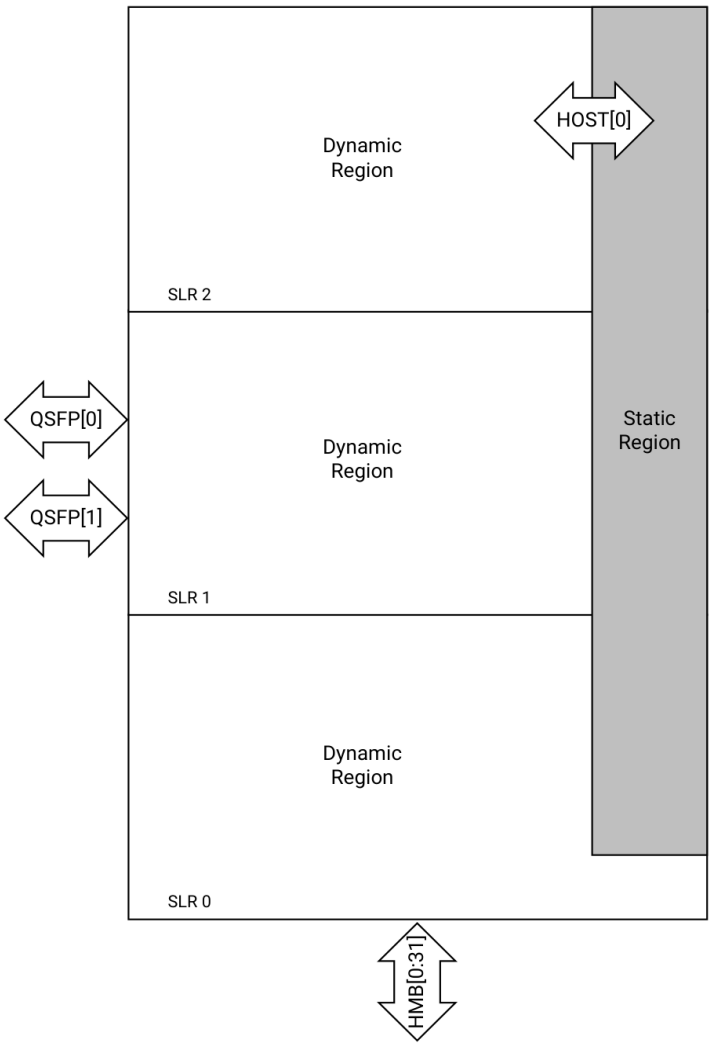


Figure 2.6: from [6]; Floorplan of an AMD Alveo U55C device

chip size (see Figure 2.6). Each individual die forms what is referred to as a Super-Logic Region (SLR). SLRs can be said to almost act like separate FPGAs, in the same chip.

Designing for an SSI device requires a more careful floorplanning than with conventional FPGAs. Firstly, data connections between SLRs - called SLR crossings - pose their own set of potential problems. The chip only features a limited amount of traces that connect (vertically) adjacent SLRs, called Super-Long Lines (SLLs) by AMD/Xilinx, which additionally impose a latency penalty. No connections exist between non-adjacent SLRs. Cascadable primitives like DSPs and BRAM or UltraRAM can not be cascaded across SLR boundaries. Furthermore, different SLRs usually require separate clock buffers, which necessitates a form of clock skew correction when implementing SLR crossings. One solution is to explicitly serialize and re-synchronize data connections at SLR crossings, as is done in [7]. Additionally, Figure 2.6 depicts that peripheral I/O interfaces of the FPGA (like memory or QSFP connectors for serial communication) are tied to one specific SLR, which significantly affects placement of associated fabric logic.

Architecturally, Alveo devices, in contrast to other FPGAs, consist of a so-called “static region” and a “dynamic region”. The static region is a mostly hard-IP block with hardware for application-independent system tasks - like PCIe communication, and elementary clock generation. The dynamic region holds the classical FPGA fabric.

Concluding, the FPGAs on Alveo cards are designed to feature large amounts of highly capable computing and logic resources. However, fully leveraging these resources comes at a considerable increase in design complexity, compared to “standard” (non-SSI) FPGAs.

Operation and Host Integration

Alveo accelerators are typically programmed via Xilinx’s Vitis compiler toolchain, which wraps a Vivado hardware build flow, and controlled via Xilinx’s OpenCL-based Xilinx Runtime Library (XRT). In addition to performing the hardware build for the user kernel(s), the Vitis compilation flow infers a “shell” on the FPGA, whose foremost tasks are PCIe communication, providing the kernel with generic AXI4 or AXI4-Lite interfaces to PCIe and global memory channels, and setting up host application Direct Memory Access (DMA). The shell can optionally include clock management. Resource-wise, one part of the shell resides in the static region of the chip, while the other part (memory interfaces, for instance) resides in the dynamic region, and hence consumes fabric logic resources. The static region is exclusively managed by the toolchain. Eventually, the toolflow embeds the user kernel(s) into the shell.

It is of course also possible to program the card in the “conventional” plain Vivado way, without the Vitis toolchain. However, in that case, the XRT framework can no longer be used to interface with the chip. All FPGA interaction in this thesis is done via Vitis toolchain and XRT libraries.

2.3. Fundamentals of Quantum Computing

This section describes the necessary fundamental principles and techniques for handling quantum state vectors, with respect to this thesis. The section explicitly does not aim at providing a complete introduction into Quantum Information fundamentals, nor does it provide all mathematical depth and cohesion of a thorough introduction. For an in-depth discussion, the reader is referred to the textbook by Nielsen & Chuang [8]. Since the objective is supporting this thesis, several otherwise elementary topics in Quantum Information are left out, such as universal gate sets, and any effects related to specific physical qubit implementations.

The following sections are subdivided as follows. Section 2.3.1 introduces the notions of quantum systems and quantum state vectors, explains operations on single- and multi-qubit quantum state vectors, and introduces superposition and entanglement as part of the process. Section 2.3.2 focuses on aspects related to non-ideal (or noisy) qubits and gate operations, including the introduction of quantum state fidelity. Finally, Section 2.3.3 gives an overview on typical computational applications of Quantum Information, with a clear distinction between Quantum Computing and Quantum Network scenarios.

2.3.1. Quantum Information and State Vectors

Qubits, Quantum State Vectors and Superposition

Quantum-based computation revolves around *qubits* as the essential unit of Quantum Information, which represents the building block of many systems and algorithms - analogous to the bit, in conventional

computing systems. One or multiple qubits form a *quantum system*, and a quantum system in turn always is in a certain state, its *quantum state* (which can be manipulated). In contrast to classical bits, qubits have the ability to exist in a *superposition* of two states, with associated probability amplitudes - where classical bits (logically) can only be either 0, or 1, at a time. A qubit collapses into one of the superposed states, upon observation. In conventional Computer Engineering, the behavior upon observation is similar to a CMOS inverter whose input is held at $V_{DD}/2$, and then disconnected - which would let the output appear as either 0, or 1, with relatively equal probabilities. This type of probabilistic action however only relates to the physical implementation of a bit, not to the mathematical (binary) concept of a bit. Superposition in contrast is not an implementation-dependent phenomenon, it is instead part of the mathematical description of qubits.

An often used mathematical formalism to describe a quantum state (and the only formalism that is relevant in this thesis) is a *quantum state vector*, which are vectors in a finite dimensional complex vector space $\mathbb{C}^d$. They represent a system of n qubits as a normalized vector, whose entries are also called *amplitudes*, in a basis of 2^n orthonormal quantum states. A state vector is typically given in the so-called "bra-ket" (or Dirac) notation, such that the state vector for a qubit α looks as follows:

$$\text{“ket } \alpha\text{”:} \qquad |\alpha\rangle \coloneqq \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \quad \alpha_1, \alpha_2 \in \mathbb{C}$$

Per convention:

$$\langle\alpha| \coloneqq |\alpha\rangle^\dagger = \begin{pmatrix} \alpha_1^* & \alpha_2^* \end{pmatrix}$$

The amplitudes denote the probability for the quantum system to be in the respective basis state:

$$p\left(|\alpha\rangle = |i\rangle\right) = |\alpha_i|^2 \tag{2.1}$$

Accordingly, "normalization" originates from the fact that probabilities must always add up to 1, thus

$$\sum_i |\alpha_i|^2 = 1 \tag{2.2}$$

The orthonormal basis typically consists of one pair out of the so-called *cardinal states*, which are defined as $|0\rangle \,/\, |1\rangle$, $|+\rangle \,/\, |-\rangle$, and $|+i\rangle \,/\, |-i\rangle$. Their relation is explained in appendix Section A.1.1. The exact implementation of $|0\rangle \,/\, |1\rangle$ depends on the physics of the qubit, and is not of further importance in this thesis. The basis $|0\rangle \,/\, |1\rangle$ is also referred to as the *computational basis*. Per convention, in computational basis, it holds that

$$|0\rangle \coloneqq \begin{pmatrix} 1 \\ 0 \end{pmatrix} \;,\; |1\rangle \coloneqq \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Theoretically, state vectors can be given in any orthonormal basis, and can be transformed from one basis into another one. However, for the remainder of this thesis, state vectors are always given in computational basis, if not explicitly stated otherwise.

1-qubit state vectors can additionally be visualized in a 3-dimensional representation, the *Bloch sphere*, as shown in Figure 2.7. The figure visualizes the state

$$|\psi\rangle = \psi_1 |0\rangle + \psi_2 |1\rangle = e^{i\gamma} \left(\cos(\frac{\theta}{2} |0\rangle) + e^{i\phi} \sin(\frac{\theta}{2} |1\rangle) \right)$$

Although the vector theoretically has 4 degrees of freedom (2 complex numbers), the Bloch sphere only offers 2 degrees of freedom (ϕ and θ), because γ can be extracted from the vector as a global phase, and due to the normalization condition (which mathematically stems from the purity condition). The global phase of a qubit can not be measured, and is irrelevant for quantum operations. It therefore is omitted in the Bloch sphere. The normalization condition dictates that the vector has length 1, hence the sphere. The Bloch sphere provides a good depiction of how a state vector can be projected onto an axis, which is formed by 2 orthonormal basis states, in order to obtain the probability for the quantum

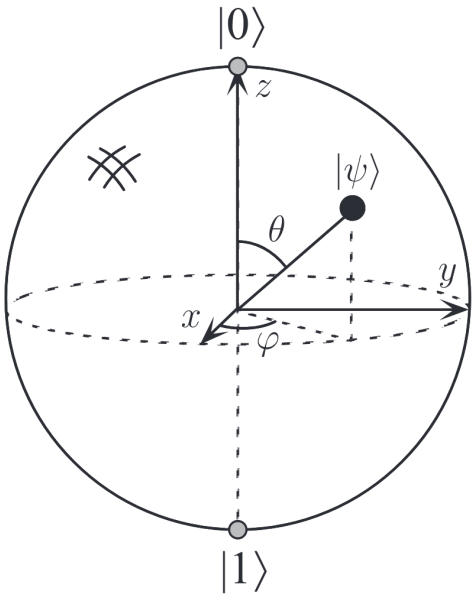


Figure 2.7: from [8]; The *cardinal single-qubit states* of the computational basis, and an arbitrary state $|\psi\rangle$, displayed on the Bloch Sphere.

system to collapse into a certain basis state. The cardinal states here lie at the intersections between the (negative and positive) x-, y- and z-axis with the sphere, respectively.

When expressing a multi-qubit system in one state vector, the orthonormal basis states are formed by the tensor product of the basis states of the individual qubits. For a 2-qubit state vector in computational basis, the basis would be $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. For this reason, a state vector representing an n-qubit system has 2^n entries. Independent quantum systems (in quantum state vectors) can be expressed as one quantum state system, by forming the kronecker product, also called tensor product, between their state vectors:

$$|\psi\rangle = |0\rangle \quad , \quad |\phi\rangle = |1\rangle \tag{2.3}$$

$$|\Psi\rangle = |\psi\rangle\,|\phi\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} = |01\rangle \tag{2.4}$$

Equation (2.4) additionally demonstrates how the bra-ket formalism facilitates calculations on quantum states, in that it reduces a vector tensor product to concatenating ket notations, in this instance.

Single-Qubit Quantum Gates

Operations on qubits are performed through so-called quantum gates, analog to gates in classical logic circuits. In the state vector formalism, a gate is expressed as a unitary matrix (and therefore is reversible), which is multiplied with the state vector:

$$|\psi\rangle \coloneqq \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \qquad X \coloneqq \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$$

$$\begin{aligned} X \cdot |\psi\rangle &= \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix} \cdot \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} \\ &= \frac{1}{\sqrt{2}}\begin{pmatrix}-1\\1\end{pmatrix} = \frac{1}{\sqrt{2}}\left(-|0\rangle + |1\rangle\right) = \frac{1}{\sqrt{2}}\left(|1\rangle - |0\rangle\right) \end{aligned}$$

The X gate from the example is an elementary quantum gate, and part of a group called *Pauli* gates. A list of all elementary quantum gates can be found in the appendix, in Section A.2.1.

In theory, any unitary rotation matrix can be a quantum gate. On the Bloch sphere, this translates into the fact every possible single-qubit quantum gate is a rotation of the state vector about a certain axis.

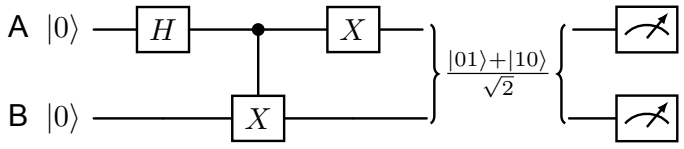


Figure 2.8: Example quantum circuit to set up 2 qubits in the $|\psi^+\rangle$ Bell state, and afterwards measure both qubits

The Pauli gates are rotations about the Bloch sphere's x, y, and z-axis, respectively. The X gate, for instance, performs a rotation by $180°$ about the y-axis. It thus transforms $|0\rangle$ into $|1\rangle$, and vice versa (as seen in the example above), and therefore has similarities to the operation of an inverter on a classical bit.

From an engineering perspective, there is an important difference in the realization of physical qubits and quantum gates, compared to the relation between conventional bits and gates. A classical gate is an always-present physical structure, through which the information bit passes. In quantum systems however, that relation could be described as reversed, in that the qubit is the component that is at a fixed physical "location" in the quantum chip. Applying a quantum gate then means applying a certain type of stimulation signal to the physical qubit (electric wave, light wave, depending on the qubit implementation).

Qubit Measurements

The exact state of a qubit or quantum system is impossible to observe (with a single measurement), due to the nature of quantum mechanics. A qubit can only be measured, in a certain basis, in which case it "collapses" into one of the respective orthonormal basis states. The two possible measurement outcomes are denoted as 0 and 1, while conventions vary as to which outcome corresponds to which state. For the remainder of this thesis, in computational basis, measurement outcome 1 corresponds to post-measurement state $|0\rangle$, and outcome 0 corresponds to $|1\rangle$. The amplitudes, as described above, determine the probability of the qubit collapsing into one or the other basis state.

Mathematically, the quantum measurements in this thesis are projective measurements. Here, a quantum state is projected onto a *projector* P_m, corresponding to one of the outcomes of an observable M (a 2x2 hermitian matrix, for single qubits). In computational basis, the projectors are

$$P_1 = |0\rangle\langle 0| = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{2.5}$$

$$P_0 = |1\rangle\langle 1| = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.6}$$

The probability for a measurement outcome m, for a state $|\psi\rangle$, is given by

$$p(m) = \langle\psi| P_m |\psi\rangle \tag{2.7}$$

It can quickly be seen how Equation (2.7) resolves into Equation (2.1), when applying any of the projectors from Equation (2.6). The result is either $|\psi_1|^2$, or $|\psi_2|^2$ (given a 1-qubit state), depending on m.

Multi-Qubit Gates and Entanglement

So far, only single-qubit gates were discussed, applied to a quantum system of one qubit. The following paragraphs focus on operations and effects in multi-qubit systems. The discussion entails single-qubit operations in multi-qubit systems, multi-qubit gates, and entanglement. These concepts will be explained along a simple, common quantum circuit - the creation and measurement of a specific so-called *Bell state*, or *EPR pair* (see also appendix Section A.1.2). The circuit is depicted in Figure 2.8.

The quantum circuit consists of two qubits (A and B), with gates applied to them in the order they appear from left to right. Firstly, A and B are both set up as

$$|\Phi\rangle_A = |\Phi\rangle_B = |0\rangle$$

Φ acts as a placeholder here, to denote a general quantum state. The first operation is an H gate (an elementary gate), applied to qubit A. Up to this point, it was sufficient to describe A and B as independent quantum systems, and therefore as two 2-element state vectors. After applying H,

$$|\Phi\rangle_A = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$

, and $|\Phi\rangle_B$ is still $|0\rangle$.

The next gate is what is called a *controlled gate*. In this case, it is a Controlled-X gate, which is also called CX, Controlled-NOT, or CNOT. Controlled gates have a number of *control qubits*, denoted by a black dot in the circuit diagram, and one *target qubit*, showing the operation that is conditionally performed. The indicated operation is applied to the target qubit if and only if **all** control qubits are $|1\rangle$. The state of the control qubit(s) is never affected. Since the control qubit can be in a superposition state, controlled gates can cause the target qubit to end up in a superposition state, even if before it was not. The standard controlled gates can also be found in appendix Section A.2.2.

In order to apply the controlled gate, all involved qubits must be expressed as one quantum system. Therefore, for the CNOT gate:

$$|\Phi\rangle_{AB} = |+0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$

In ket notation:

$$|\Phi'\rangle_{AB} = \text{CNOT} \cdot \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

In matrix notation:

$$|\Phi'\rangle_{AB} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The next operation is an X gate, applied to qubit A. Since the X gate is a 2x2 matrix, which eventually needs to be multiplied with a 4-element state vector, the gate matrix must be adapted. This is done by virtually applying an *identity gate* (with the identity matrix as gate matrix) to all other qubits. Analog to combining quantum state vectors via a tensor product, concurrent gate operations also need to be combined via a tensor product (in the correct order):

$$X_A = X \otimes \mathbb{I}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Accordingly, X_B (X applied to qubit B), would be formed by computing $\mathbb{I}_2 \otimes X$.

For the remainder of this thesis, the distinction between X_A (or X_B) on the one hand, and X on the other hand (or any other operation matrix) is extremely important. Therefore, the terms *base gate matrix* and *full operation matrix* are introduced here for this thesis specifically. The base gate matrix is the single-qubit 2x2 matrix of a gate, the full operation matrix is the result of the base gate matrix, and the tensor products with identity matrices that extend it to the full space (from both left and right side). As a short term, it will be said in this thesis that the base gate matrix is *extended* (by identity matrices) into the full operation matrix.

The intermediary state resulting from the extended X gate in the example circuit is only given here in ket notation:

$$|\Phi''\rangle = X_A |\Phi'\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

The final step consists in measuring the two qubits. Here, the measurements will be done one qubit at a time, starting with qubit A. The result is mathematically equivalent to measuring both qubits at the

same time. The measurement outcome probability can be described as an expectation value, or by giving the probability for one measurement outcome (since there are only two possible outcomes for a single qubit, and the probabilities must add up to 1). The latter is chosen here, because it is the method that is used later on in the thesis.

The probability for qubit A to collapse into $|0\rangle$ can again be determined either via ket formalism, or in vector notation. The argumentation in ket formalism is that the quantum system is in a superposition of the states $|01\rangle$ and $|10\rangle$, both with amplitude $\frac{1}{\sqrt{2}}$. In the superposed states, qubit A is $|0\rangle$ in the first case ($|01\rangle$), and $|1\rangle$ in the second case. Due to Equation (2.7), the probability of A to collapse into $|0\rangle$ is the sum of all squared amplitude absolute values in the quantum system that contains A, where A is $|0\rangle$. In this case therefore:

$$p(A = |0\rangle) = \left|\frac{1}{\sqrt{2}}\right|^2 = 0.5$$

The same result can be obtained by applying Equation (2.7) to the vector notation. In order to do that, the projector needs to be "extended" in the same way that base gate matrices in multi-qubit systems are:

$$P_{A=|0\rangle} = P_{|0\rangle} \otimes \mathbb{I}_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.8}$$

$$\begin{aligned} p(A = |0\rangle) &= \langle \Phi'' | \, P_{A=|0\rangle} \, |\Phi''\rangle \\ &= \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = 0.5 \end{aligned}$$

Therefore, measuring A yields outcome 0 or 1, both with $50\,\%$ probability.

An important observation is made when discussing the measurement of B, after measuring A. As shown, $|\Phi''\rangle$ only has 2 options to collapse into, upon a measurement of qubit A. If A is measured as $|0\rangle$, $|\Phi''\rangle$ has collapsed into $|01\rangle$. If A is measured as $|1\rangle$, $|\Phi''\rangle$ has collapsed into $|10\rangle$. In both cases, there is no need to measure qubit B, in order to know the outcome. All possible options for $|\Phi''\rangle$ to collapse into, given a certain measurement outcome for qubit A, yield the same outcome for qubit B.

The reason (in this case) for the measurement outcomes of A and B to depend on each other lies in the fact that, in the term for $|\Psi''\rangle$, they can not be "separated". In comparison, $|\Phi\rangle$ before the CNOT gate could be expressed as follows:

$$|\Phi\rangle_{AB} = \frac{1}{\sqrt{2}} \left(|0\rangle + |1\rangle\right) \cdot |0\rangle = |\Phi\rangle_A \cdot |\Phi\rangle_B$$

Such a decomposition into a term with separate states $|\Phi\rangle_A$ and $|\Phi\rangle_B$ is not possible for the intermediary states after the CNOT gate, and hence for $|\Phi''\rangle$. In terms of terminology, before the CNOT gate, A and B are said to be *separable*, whereas after the gate, they are not. If (and only if) qubits are not separable, they are *entangled*.

In the above example, after determining a measurement result, the state vector needs to be modified to represent the new, post-measurement state. When measuring qubit A, for instance, it automatically becomes separable again (for the measurements used in this thesis), because all states that do not comply with the measurement result for qubit A are eliminated from the superposition. Therefore, after the measurement, the new state can be described as a quantum system with or without qubit A. The two options have various names in the literature, like "in-place/not in-place" measurement, or "destructive/non-destructive" measurement. This thesis will mostly use "destructive/non-destructive" measurement. Mathematically, a destructive measurement "traces out" the measured subsystem. For a state vector, that means removing all vector amplitudes that do not correspond to the measurement result - thereby halving the vector size. In non-destructive measurements, they are kept, but set to 0.

In both cases, the vector needs to be normalized again, by dividing the remaining amplitudes by the square-root of the probability for the eventual measurement outcome:

for measurement outcome: 1 / $|0\rangle$ (although irrelevant here since $p = 0.5$)

$$p(0) = \frac{1}{2}$$

destructive measurement:

$$|\Phi'''\rangle_B = \frac{\frac{1}{\sqrt{2}}|1\rangle}{\sqrt{p(0)}} = |1\rangle$$

non-destructive measurement:

$$|\Phi'''\rangle_{AB} = \frac{\frac{1}{\sqrt{2}}|01\rangle}{\sqrt{p(0)}} = |01\rangle$$

Thesis-specific Terminology

To conclude the discussion on Quantum Information and quantum operations, this section introduces the notion of *monomial gates*. The term is defined specifically for this thesis, because these gates play an important role in the computation scheme proposed in Chapter 3.

A monomial matrix is a matrix that has exactly one non-zero element in any row and colum. Accordingly, monomial gates are quantum gates with a monomial operation matrix (in the computational basis). It is irrelevant whether the base gate matrix or an extended full operation matrix is considered, because the full operation matrix is monomial if and only if the base gate matrix is monomial (proof in the appendix, Section B.1.1). Monomial and non-monomial gates can therefore immediately be differentiated by their base matrices. Except for the H gate (and with it, any "general rotation" gate), all standard single-qubit and two-qubit quantum gates (see appendix, Section A.2.1) are monomial gates.

Quantum gate matrices in particular have the characteristic that all (standard) non-monomial quantum gates have at most two non-zero entries per row and column. That is because there are only single-qubit non-monomial gates, and thus only 2x2 non-monomial base gate matrices, while the "gate extension" via identity matrices preserves the number of non-zero entries per row and column (proof in Section B.1 in the appendix). Therefore, all standard quantum gate matrices can be treated as either monomial gates, or as single-qubit gates with 2x2 base gate matrices (with several entries potentially being 0). This thesis uses the term *dense gates* for gates that are treated as having 2x2 base matrices - due to the densely populated base gate matrix - such that monomial gates and dense gates collectively cover all appearing quantum gate operations.

2.3.2. Noise and Fidelity

In the above discussion, qubits were considered ideal, meaning that they would always retain their state over time, and gate operations were considered to be executed perfectly. Even more than with real classical bits, that is rarely the case with real quantum bits. Physical qubits themselves are relatively unstable (in current technology), and real quantum gates do not always perfectly perform the rotation that was intended. Additionally, just like classical channels, quantum channels do not always perfectly transmit a qubit. All these effects contribute to "noise" in quantum systems. The following paragraphs give a brief overview on noisy quantum systems, and afterwards introduce the *fidelity* as a measure for closeness, or "correctness", of quantum states. However, the discussion is kept relatively high-level, because this thesis only requires a general understanding of noisy quantum systems and fidelity, but no mathematical knowledge on specific noise models.

Typically, noise is expressed in the so-called *density matrix* formalism, instead of state vectors. However, this work's proposed computation scheme and hardware design are not defined for density matrices yet (see outlook in Section 9.2.2). Therefore, apart from a short motivation in this and the following paragraph, density matrices are omitted in this section. Density matrices represent quantum systems in 2-dimensional (complex) matrices, instead of 1-dimensional vectors. A state vector can always be transformed into a density matrix, without loss of information, but not vice versa. In terms

of computation, this thesis does not require exact knowledge about performing quantum gates or measurements on density matrices. It is sufficient to know that density matrices require a similar algebraic "framework" to state vectors (matrix-matrix multiplications, inner products), accounted for the additional dimension. Further information can be found in [8].

The additional dimension in density matrices allows expressing so-called *mixed quantum states*. These occur when an entire quantum system can be in one of several different states. Mixed states are not to be confused with superposition, the two are independent concepts. A qubit can be in a mixed state, of multiple superpositions. The opposite of a mixed state is a *pure state*, which is what state vectors are restricted to. This is the reason that every state vector can be transformed into a density matrix, but not the other way around. Mixed states are useful when studying noisy quantum systems, as the following paragraphs will show. They provide an analytic, non-probabilistic way to describe the effects of various noise types on a quantum system, which is not possible with state vectors.

Noise Models

Physical qubits experience a number of phenomena, which cause them to degrade over time. These can be described as noise models, or error models. Several effects are mostly related to qubits in a quantum chip or register ("in memory", in classical computing terms), others mostly appear in the context of transmitting a qubit via quantum channels. The distinction however is not always strict. Additionally, gate operations can introduce noise. As a collective term for qubit degradation, that is not related to gates, the qubit is said to *decohere*.

Qubits in a quantum register often experience *relaxation* and *dephasing*. Relaxation appears when a qubit loses energy to the environment, which makes it "drop" from the higher-energy $|1\rangle$ state to the lower-energy $|0\rangle$ state. An analogy is a classical bit in DDR memory, which loses its charge over time, and therefore can drop to bit value 0. Dephasing causes a qubit to lose the phase coherence between its $|0\rangle$ and $|1\rangle$ state, which is modeled by randomly applying a Z gate, with a certain probability.

Both mechanisms discussed so far illustrate the added value of mixed states, and hence density matrices. A mixed state can describe a system where relaxation or dephasing may or may not have happened, alongside with the respective probabilities. Given a state vector, or pure state, in contrast, the Z gate, modeling the relaxation, is either applied or not. In any case, the state vector is left in one of the two pure states, with the other option "discarded". In other words, density matrices can analytically express what state vectors can only yield by stochastically calculating the same quantum circuit multiple times, assuming perfect probability distribution.

When transmitted via quantum channels, qubits mostly experience depolarization, bit-flipping, and also dephasing. Depolarization is best described as "randomization" of the qubit. It causes the qubit to decay into a random pure state. With density matrices, this means decaying into the maximally mixed state. Bit-flipping is similar to classical bit flipping, it is the random inversion of a qubit/bit - which is equivalent to an inverter for classical bits, or an X gate for qubits. Additionally, quantum channels can cause total loss of a physical qubit (for instance in the form of a lost photon), which analytically is beyond the scope of this section.

Lastly, any quantum gate is prone to be noisy, namely if the gate rotation is not executed perfectly. From an electrical engineering perspective, that has a rather intuitive visualization. Quantum gates are mostly wave stimulation signals (electrical or light), which are applied over a certain period of time, at a certain frequency, and at a certain amplitude. If any of those parameters is slightly incorrect, that might for instance lead to a non-exact rotation angle, or rotation axis (in the Bloch sphere). Noisy gates are different from the other noise models, in that their effect is not modeled by discrete actions (like the Z gate in relaxation, which is either applied or not), but rather in the form of a "continuous variation" (misalignment in rotation axis or angle can be a continuous distribution).

In summary, various types of physical effects, mathematically represented by noise models, cause time-dependent qubit decoherence. State vectors do not have a means to analytically express the effects of noise, they can only rely on stochastic evaluation. Density matrices allow analytical modelling of noisy quantum memories and channels, via mixed states.

Fidelity

Given that quantum states can be noisy, as demonstrated, one requires a measure to determine the "correctness" of a quantum state. Comparing state vectors entry-per-entry does not yield meaningful results, due to the global phase (Section 2.3.1). Therefore, *fidelity* is used instead. For **pure** states $|\phi\rangle$

and $|\psi\rangle$, fidelity is defined in [8] as the following inner product:

$$F(|\phi\rangle, |\psi\rangle) = |\langle\phi|\psi\rangle| \tag{2.9}$$

Other definitions use the "squared fidelity":

$$F(|\phi\rangle, |\psi\rangle) = |\langle\phi|\psi\rangle|^2 \tag{2.10}$$

All fidelities in this thesis however are given using Equation (2.9), which sometimes is also called "square-root fidelity". Both forms have a minimum value of 0 (orthogonal states), and a maximum value of 1 (equivalent states).

The fidelity between two states has in fact already been used in this section, when defining the likelihood for a measurement outcome in Equation (2.7). The projector P_m here acts as a density matrix that represents the pure state corresponding to measurement outcome m, while the squared fidelity between a pure state $|\psi\rangle$ and a density matrix ρ is defined as

$$F(|\psi\rangle, \rho) = \langle\psi| \rho |\psi\rangle \tag{2.11}$$

Therefore, the (squared) fidelity can also be interpreted as the likelihood of one state to behave as a certain other state, upon measurement.

2.3.3. Applications

The effects described in the above section can be leveraged for both computing purposes, and for information transferring tasks - referred to as Quantum Computing, and Quantum Networks (or Quantum Communication). As a collective term for any computational application based on Quantum Information - among which Quantum Computing and Quantum Networks - the remainder of this thesis will use *Quantum Computation*. While both fields utilize the same principles, they differ significantly in the typical algorithms and sizes of entangled quantum states. Additionally, they both take advantage of different characteristics of quantum effects. Therefore, the following paragraphs treat Quantum Computing and Quantum Networks separately, giving brief overviews on either application field. The discussion is rather concise, because this thesis generally does not require knowledge about specific algorithms, with very few exceptions (that are explained in this section). The focus lies on understanding general usecase scenarios, and on the differences in quantum system composition.

Quantum Computing

Quantum Computing algorithms aim to leverage qubit entanglement for performing computation, with the goal of providing exponential reductions in computation time, compared to specific classical algorithms (the computation time of a quantum algorithm is measured in the number of gate operations). One (technically debatable) visualization is that superposition, alongside with exponential information capacity due to entanglement, allows a quantum computer to explore an exponential number of solutions "at the same time", hence the reduction in runtime complexity. Several sources refer to this phenomenon as *quantum parallelism*. It is vital to understand that, generally, the applicability of quantum computers is restricted to **specific** problems, for which an efficient quantum algorithm has been found. From a computer architectural point of view, these quantum computers therefore function as a type of accelerator, not as a general-purpose computer.

Meaningful Quantum Computing algorithms consist of the same building blocks as the Bell state generator from the previous section (Figure 2.8). A set of qubits is first set up to a known initial state. Then a set of gates creates full entanglement across the quantum system (H and CNOT, in the example), followed by a series of gate operations, to achieve certain semantics (X in the Bell state generator, to transform one Bell state into another). Finally, qubits are measured - because that is the only method of observation in a quantum system - and the results interpreted. This section will not go into more circuit-level detail on Quantum Computing, since it is not required for this thesis.

In order to understand the characteristics of quantum states in Quantum Computing algorithms, two of the most famous and most impactful algorithms are studied as examples. These are **Shor's Algorithm** [9] and **Grover's Algorithm** [10]. Grover's algorithm is a search algorithm, which allows reducing the complexity of an unstructured search among N elements from $\mathcal{O}(N)$ to $\mathcal{O}(\sqrt{N})$. Shor's algorithm performs prime factorization of large integers, in polynomial runtime - while the best known

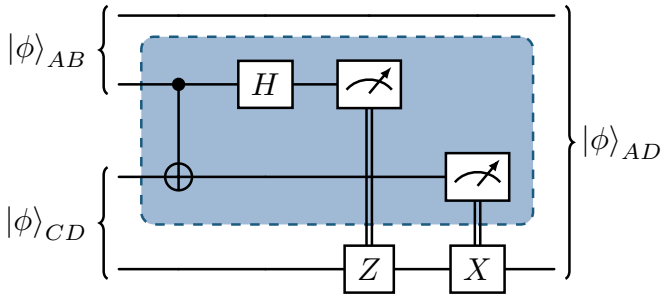


Figure 2.9: *Entanglement Swap*, the essential circuit in a *Quantum Repeater*

classical algorithm has (sub-)exponential runtime. Therefore, Shor's algorithm provides a polynomial-time method to break any cryptography scheme that relies on the complexity of prime factorization, among which RSA.

Both presented algorithms have in common that their problem size is restricted by the number of (entangled) qubits - which is the reason that they are not technically feasible yet, at a meaningful scale. Grover's algorithm, with n qubits, can only search among $N = 2^n$ elements. For Shor's algorithm, in order to factor the product of two n-bit prime numbers, it is considered to need $(2 \cdot n)$ qubits for reliable results (while $n + 1$ is a theoretical lower boundary). Therefore, consistently breaking common $2048\,\mathrm{bit}$ RSA keys would require 2048 entangled (ideal) qubits. The common characteristic of both examples, and Quantum Computing algorithms in general, is that they rely on one entangled state, with a large number of qubits.

Quantum Networks

Quantum Networks aim to connect distant nodes at a quantum level, using qubits, and so-called "quantum channels" (alongside with "classical" information channels). The main advantages lie in information security and transmission speed. As an example, **Quantum Key Distribution (QKD)** allows establishing a cryptographic key that is physically impossible to eavesdrop on. Due to the nature of quantum mechanics, a qubit can not be cloned or duplicated (*No-Cloning Theorem*). Therefore, a quantum channel can not be simply "intercepted" like a classical channel. Additionally, QKD uses physical effects to provide guaranteed detection of any eavesdropping attempt. For this reason, the communication is cryptographically protected by laws of physics, instead of computational hardness assumptions. Another usecase for Quantum Networks is linking multiple quantum chips for distributed Quantum Computing, in order to increase the number of entangled qubits. For this thesis, a non-strict distinction will be made between "Quantum Networks", and "Quantum Networking". "Quantum Networks" here denotes raw physical linking between quantum nodes, while "Quantum Networking" refers to applications that are carried out on Quantum Networks - such as QKD or distributed Quantum Computing. For further techniques and usecases in Quantum Networking, like quantum teleportation and clock synchronization, the reader is referred to common literature. They are not required for this work.

Analogous to Quantum Computing, the typical quantum system involved in Quantum Networking is illustrated by studying a common algorithm. *Entanglement swap*, the foundation of any *quantum repeater* (Figure 2.9), is a technique that allows to create an entangled qubit pair between two distant nodes, with a repeater station in the middle. It is therefore imperative in creating any quantum link between separate nodes, regardless of the Quantum Networking application. The circuit in Figure 2.9 is explained in this paragraph functionally, but not mathematically. States AB and CD represent nodes at a physical distance, which both hold the same EPR pair ($|\phi^+\rangle$, for instance), of which they send one qubit to the repeater - denoted by the blue box. The repeater performs a Bell measurement (CNOT and H gate, followed by measuring both qubits), and sends the measurement results to node CD (or AB). If the first measurement outcome was 0 (corresponding to $|1\rangle$), a Z gate is applied to D, and likewise with the second measurement outcome, and an X gate. Afterwards, the quantum system AD is guaranteed to be in the same Bell state than AB and CD at the start, and therefore a fully entangled link between the two distant nodes has been established.

The important characteristic in the quantum repeater, in terms of quantum states, is that the largest entangled quantum system to appear in the algorithm (after the CNOT gate) consists of 4 qubits. Even

when employing multiple repeaters in series, between 2 nodes - called a *repeater chain* - the maximum state size remains 4 qubits. A 2-node link is sufficient for many important schemes, like QKD It is intuitive however that many networks consist of a multitude of nodes. The conclusion is that many Quantum Networking applications involve a high number of relatively small states, which are dynamically entangled with each other, and reduced in size again by measurements. This is in stark contrast to typical Quantum Computing applications, which operate on only one state, of practically constant size, with a high number of qubits.

2.4. Simulating Quantum Computation

Simulating quantum algorithms (both Quantum Computing and Quantum Networking) plays an important role in the evaluation of quantum applications, because (general) quantum hardware is not feasible yet at a “meaningful” scale. On the one hand, the largest current Quantum Chips (for general Quantum Computing) have a little over 1000 qubits [11]. However, in terms of noisiness and error rates, these are far away from behaving like “ideal” qubits. Therefore, in order for these qubits to become usable in Quantum Computing applications, it is necessary to introduce so-called *quantum error correction (QEC)*. QEC will not be discussed in detail here, because it is not directly relevant for this thesis. The concept consists in logically combining a number of *physical* qubits into one *logical* qubit, thereby introducing redundancy, which allows schemes to correct faulty qubits.

In current technology, 100 and more physical qubits and more are considered necessary to form 1 logical qubit. Accordingly, an application like Shor’s algorithm, for breaking $2048\,\mathrm{bit}$ keys, would require a quantum computer of roughly $200{,}000$ qubits, or more. For Quantum Networks, physical point-to-point links have been established between two nodes at several kilometers distance [12]. However, scaling up the technology to larger distances, and a large number of nodes, remains ongoing research. Simulating quantum computation allows researchers to evaluate scenarios that are not feasible yet with existing technologies. Potential usecases are performance elaboration of specific algorithms, studying the impact of noisy electrical components on a larger system, and validating design approaches.

Since, as explained in Section 2.3.2, (noisy) quantum computation simulation is practically always a stochastic process, it is common practice for simulators to perform the same simulation multiple times. The outputs are observed, and analyzed either in terms of average outcome, or in terms of a certain success probability, depending on the application.

2.4.1. Classification

This section gives a brief introduction to the existing types of quantum computation simulators, and their key differences. The thesis does not require knowledge of particular quantum simulators, with the exception of NetSquid, for which additional detail is provided in Section 2.4.3. The goal of this section is to convey an understanding of the heterogeneity in simulator approaches, which translates into design decisions at several points during the thesis.

As indicated in Section 2.3.3, there is a clear difference in operation flow between Quantum Computing and Quantum Networking. Accordingly, most simulators are at least designed for one of the two scenarios, if not only applicable to one scenario. For instance, since Quantum Computing applications have no need to join quantum states at any point, many Quantum Computing simulators are entirely unequipped to operate on multiple states, or combine them - which makes them unusable for simulating Quantum Networks.

Next to differences in functionality sets, simulators can be implemented with different targets in mind. If a simulator is designed for accessibility to a wide user base, it is likely to feature a generic user interface, and be written in a potentially less performant, but widely known language, like python. A simulator focusing on maximum execution speed on the other hand, commonly uses a more low-level, compiled language, and might shift some runtime overhead into compile time. Consequentially, it might be more difficult to program, potentially restricted in flexibility, and be less accessible to an audience without a background in computer science.

Considering Quantum Network simulators, designs have been published with a wide range of tradeoffs between low-level modularity and flexibility on the one hand, and large-scale network simulation on the other hand. Four noteworthy implementations are named and categorized in this paragraph, because their differences showcase concepts that occasionally reoccur over the course of the thesis. NetSquid [13] (more detail in Section 2.4.3) and SeQUeNCe [14] are very low-level, fine-grained

simulators. In a classical network layer model, they could be said to operate at physical and link layer level, while providing an application layer API. Both model individual components of a quantum network (channels, quantum processors, quantum registers), have an event-based operation model that includes simulation time to accurately account for time-dependent noise, and support state vectors as well as density matrices. NetSquid additionally supports two representations based on stabilizer states, which is a formalism to very efficiently express a reduced set of quantum states and operations (generality is traded for simulation efficiency). QuNetSim [15] is a more high-level simulator, that does not operate below application level. In addition, it is not event-based. Lastly, QuISP [16] is the most high-level simulator of the selection, which aims to eventually simulate up to $10{,}000$ network nodes. It achieves that scale by assuming a quantum state as known due to applied protocols, up to stochastic errors, which are tracked. Errors are the error models formulated in Section 2.3.2, except for noisy gates, which the formalism can not express. The concept bears similarities to stabilizer states, and shares the characteristic that it trades generality for computation efficiency. The first learning from this overview is that simulators employ different quantum state representation formalisms, depending on which one suits the simulation target best. Additionally, it becomes apparent that simulators vary in their execution models, which can have implications for accelerated computing.

The analogous discussion for Quantum Computing simulators takes place later in this chapter, in Section 2.5, in the context of quantum simulation on FPGAs.

2.4.2. Quantum State Vector Computation

The previous section highlighted that there is a close relation between quantum system representation, computation efficiency, and quantum simulator application field. This section provides a thorough study on computation efficiency **within** the state vector formalism - since it is the formalism that this work is based on - which takes the discussion from formalism-level, in the previous elaboration, down to implementation-level. For the remainder of this thesis, the term *(quantum) state vector computation* refers to any mathematical operation on quantum state vectors, regardless of the implementation.

Several schemes for performing state vector computation will be presented. Additionally, the emphasis will be on accelerated computing, because this work revolves around an accelerator for state vector computation. In contrast to the previous section, this section does not differentiate between Quantum Computing and Quantum Networking. Although certain schemes might have more usage in one of the two areas, the mathematical principles are the same, and all techniques discussed here **could** theoretically be used in either type of simulation.

Mathematically, the majority of operations on state vectors are matrix multiplications, as Section 2.3.1 showed. Due to the ubiquity of matrix multiplications in resource-demanding computation applications, there is a long list of optimization schemes to matrix operations - both algebraically (like matrix decompositions) and computationally (SIMD execution) - that apply to various situations, depending on the matrix structure and application. The following paragraphs first discuss approaches known from general matrix computation, before highlighting several methods specifically designed for quantum simulation. The latter class will lay the foundation for the newly proposed technique (Section 3.3).

In terms of categorization, all techniques will be analyzed with respect to accelerated computing, and DDR memory effects. Implications on DDR memory are important due to the exponential growth of state vectors in the number of qubits, which in many situations leaves DDR memory as the only feasible memory type to hold state vectors, and potentially operation matrices. In particular, the characteristics of interest are computational complexity, parallelizability, memory footprint, and memory access patterns (both in terms of sequential or random accesses, and in terms of overall locality). An overview in table form is given in a later chapter (Table 3.1), alongside with this work's newly derived scheme.

Standard Techniques - General Matrix-Vector Multiplication

General matrix-vector multiplication consists of scalar products between matrix rows and the columnar vector. Assuming a quadratic matrix (which always holds for quantum operation matrices), the number of individual multiplications grows quadratically with the vector size. However, because the multiplications are all independent from each other, the operation offers good parallelizability, although it requires a post-multiplication accumulation step per matrix row. The matrix's memory footprint is substantial, because its size grows quadratically with the vector size. Lastly, the computation allows for sequential reading and writing, concerning both the matrix and the vector. Nonetheless, the data accesses alternate between the matrix and the vector - under the assumptions of row-wise matrix processing, and no

sufficient local vector buffering resources - which reduces locality in memory accesses.

Standard Techniques - Sparse Matrix Multiplication

Due to the amount of zeros in quantum operation matrices, sparse matrix-techniques are a common approach to execute quantum operation simulations. The NetSquid simulator, for instance, offers the option to use the established Compressed Sparse Row (CSR) matrix representation for computation, which most importantly stores only the non-zero elements of a matrix, in row-wise order. For quantum operation matrices, storing only non-zero elements reduces the memory usage complexity (in the state vector size) from quadratic to linear, which follows from the proof in appendix Section B.1. The same complexity reduction holds for the computation, because multiplications "by 0" are eliminated. The independence of the multiplications, and thus the theoretical parallelizability, is not affected, compared to general matrix-vector multiplication. However, memory accesses to the state vector become random (either during read or during write), instead of sequential.

Quantum-Specific - Pairwise Gate Operation

Quantum operation gate matrices, given that they only result from a base gate matrix and tensor products with identity matrices, have a regularity to them that sparse matrix multiplication is not able to exploit. Jones et al. [17] described a significant simplification: Instead of multiplying the full gate operation matrix with the state vector, there is a decomposition of the state vector into pairs of elements (for a single-qubit base gate), such that it is equivalent to multiply the base gate matrix with the 2-element vectors formed by these pairs, and afterwards "re-assemble" the pairs into the result vector. In this thesis, this concept will be referred to as *pairwise gate operation*. The technique eliminates any need for the full operation matrix to ever be computed, or be present in memory. That saves computations, frees up substantial memory/buffering resources, and improves memory access locality, since it is no longer necessary to alternate between reading from the state vector and from an operation matrix. However, accesses to array-like state vectors in memory must be considered random, because the element pairs depend on any operation's target qubit, and thus can not be predicted.

Quantum-Specific - Labeled-Hashed Representation

With the help of pairwise gate operation, Jaques et al. [18] developed a state vector representation for maximizing the feasible simulation state size - leveraging sparsity in state vectors - which mainly targets Quantum Computing simulations. Analog to sparse matrix techniques, they only store the non-zero elements of a state vector, in pairs of {*label*, *amplitude*}, where the "label" is the element index in the vector, and the "amplitude" the value. The method leads to great reductions in memory footprint in non-noisy simulations (meaning that noisiness in quantum states, due to effects like non-ideal gate fidelity and quantum memory noise, is ignored). In order to still be able to address elements by their label, the addresses for vector elements are determined by applying a hash function to the labels. Consequently, vectors are stored in no particular semantic order, and can additionally not be assumed to occupy consecutive memory - which prevents any form of sequential memory accessing. This representation will henceforth be called *labeled-hashed* representation. This thesis will use the variable names (k, v) for label-amplitude pairs ("key" and "value"), to underline the Content-Addressable Memory (CAM) structure, but adhere to the semantically more fitting terminology "label" and "amplitude".

Conclusion

The study demonstrated that there are considerable differences between state vector computation schemes, in terms of computing efficiency, although all of these schemes theoretically provide the same amount of information (in contrast to an alternative quantum system representation). It remains to be decided per application, and per computing system, which approach is the most applicable one, or which of the presented concepts might contribute to a tailor-made scheme. However, it is evident that in many instances, there are significantly more efficient solutions than general matrix multiplication, which may even deviate from a classical "array-type" vector layout in memory.

2.4.3. NetSquid Quantum Network Simulator

Out of the four simulators introduced in Section 2.4.1, additional architectural detail is given on NetSquid, because the simulator plays a role in the later chapters of the thesis (see Section 5.1.3). The discussion however focuses on aspects that are particularly important during this thesis, instead of giving a general overview. NetSquid was developed as a cooperation of QuTech [19] at TU Delft, and TNO [20].

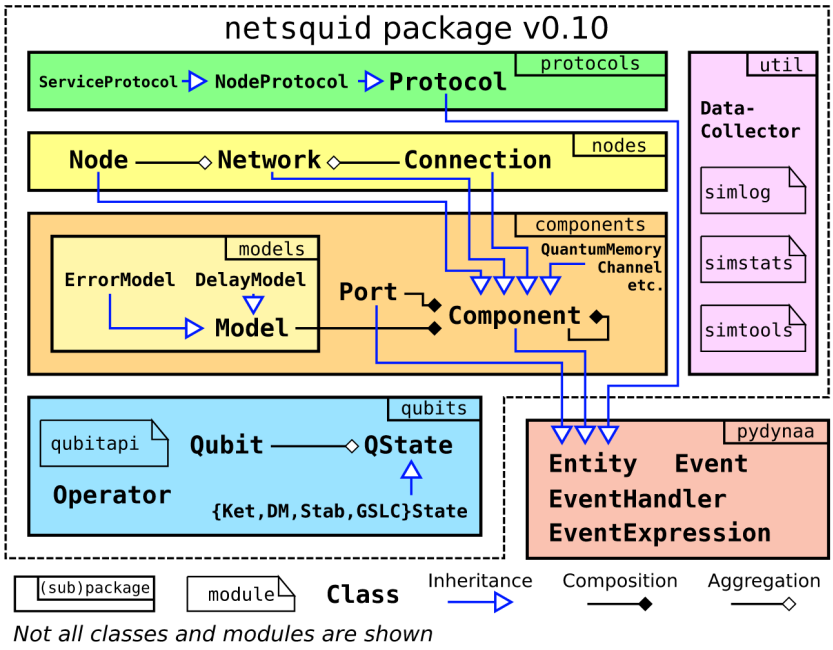


Figure 2.10: from [13]; NetSquid Quantum Network simulator functional overview. The back-end *QState* representations (*Ket,DM,Stab,GSLC*) are so-called *QRepr* objects.

Architecture and Implementation

The architecture of NetSquid is given in Figure 2.10. Logically, the packages (except for `util`) can be subdivided into user front end (`protocols`, `nodes`, `components`), quantum back end (`qubits`) and event engine (`pydynaa`). Direct user interaction only happens with the front end packages. Protocols and components act as autonomous agents, which register actions at the event engine, and receive a notification upon a result being available. Actions, that are coordinated via pydynaa, can trigger the quantum back end, by means of `Qubit` objects, which are assigned to components.

The quantum back end has a distributed, hierarchical model for qubits, quantum states, and quantum state representations. Qubits are always part of a quantum state, without knowledge about potential other qubits it is entangled with. Quantum states exist as abstract `QState` objects, which in turn are always backed by exactly one `QRepr` object. The QState object is able to bundle and address Qubit objects. The QRepr objects holds the actual state representation, and knows how to perform operations on it. Therefore, whenever an operation is performed on one Qubit, the simulator internally first resolves the Qubit into its QState, and then into the associated QRepr, to finally execute the information on the QRepr object.

Almost the entire quantum back end is based on numpy [21] and written in cython [22]. Therefore, it is compiled, performance-oriented code. The remainder of NetSquid is written in plain python, in order to provide accessibility and flexibility.

QRepr API

The QRepr class is introduced here more thoroughly, because it is central in the prototype system that was implemented during this work. It is an abstract class (also written in cython), which provides a unified Application Programming Interface (API) to support a variety of quantum state representation back ends. The full API can be found in Section 5.8.3 as part of Table 5.4, where it is discussed in the context of the developed hardware accelerator. Any quantum state formalism that NetSquid supports - state vectors, density matrices, and both types of stabilizer states - is implemented as a QRepr class. A QState object therefore is not aware of the type of underlying state representation. This allows the simulator to act modularly even in terms of quantum system representation, without affecting the behavior of quantum states as an abstract concept. QRepr additionally allows providing methods for translating between different representations, although it is up to the respective implementation whether to actually implement these methods (keep in mind that several translations are not generally possible without information loss).

Name/Authors	Hong et al.	Khalid et al.	Silva et al.	Lee et al.
Reference	[25]	[26]	[27]	[28]
Circuit	arbitrary	arbitrary at HW build time	arbitrary at HW build time	Grover's Search and QFT, applicable to other algorithms (HW build time)
Scalability	limited (parallelization unclear, bottlenecked by memory random accesses)	no (chip usage depends on quantum circuit)	no (chip usage depends on quantum circuit)	no (chip usage depends on quantum circuit)
Measurement	no	no (software)	no	not explicitly (integrated in dataflow)
Join States	no	no	no	no

Table 2.1: Functional comparison of most relevant existing FPGA implementations on Quantum Computation Simulation

2.5. FPGAs in Quantum Computing Simulation

After the previous sections introduced two large, independent topics - FPGA-based accelerated computing, and quantum computation simulation - the presented information converges in this section, which assesses the current status of FPGAs in quantum computation simulation. It is again differentiated between Quantum Computing simulation, and Quantum Networking simulation. Considering the latter, to the best of the author's knowledge, there is no FPGA-based quantum computation simulator that is capable of performing Quantum Network simulation. While Quantum Computing simulation on FPGAs is not a large field either, this study still identified a moderate level of variety in computation approaches, among published designs. Four proposed implementations are examined, comparing them in terms of functionality set and scalability. Several additional publications were assessed, but not considered in this discussion, because all relevant aspects are also found in the four examined implementations ([23, 24]). It must be stated that none of the assessed designs supports on-chip qubit measurement simulation. The implementations were mostly evaluated on the Quantum Fourier Transform (QFT) algorithm, which does not involve measurements. All designs use state vector representation exclusively for quantum systems. A comparison of the compared designs is given in Table 2.1.

Firstly, Hong et al. [25] proposed a multiplication-efficient scheme for arbitrary single-qubit and controlled quantum gates, on theoretically arbitrary state sizes (subject to main memory limitations, since they do not use on-chip memory). The concept could be said to apply pairwise gate operation, although the respective literature is not explicitly linked. At scale, the performance is hindered by random memory accessing, and the parallelizability remains unclear. All other discussed designs do not support arbitrary gate execution in hardware, without altering and re-building the FPGA design. Khalid et al. [26] and Silva et al. [27] proposed designs that physically implement an emulation of the desired quantum circuit in the FPGA, by providing the user with a gate library to specify a circuit, and afterwards re-build the bitstream. Khalid et al. do so in a Register-Transfer Level (RTL) language (VHDL), Silva et al. in contrast employ High-Level Synthesis (HLS) in C++. Due to the design concept, both approaches are limited in circuit size by the available chip resources, and the hardware usage scales with the quantum state size, thus exponentially in the number of qubits. Lastly, Lee et al. [28] presented a similar concept, with a higher level of abstraction. They separately implemented two example algorithms (QFT

and Grover's search algorithm) for FPGA, by first performing a dataflow analysis. The results were afterwards translated into a processor-like hardware structure, with a register and a parameterizable processing unit. Doing so removes any quantum computation semantics from the RTL design, which allows exploiting dataflow optimization potential - comparable to automated logic circuit optimization in Electronic Design Automation (EDA) toolchains. However, the method still causes the hardware usage to scale with the quantum state size. Due to the scaling behavior, none of the designs by Khalid et al., Silva et al., and Lee et al., was able to simulate state sizes larger than 7 qubits.

Concluding, to the best of the author's knowledge, most, if not all, relevant implementations for Quantum Computation on FPGA purely rely on "array-like" quantum state vectors, do not support quantum measurements and multiple quantum states, and are not capable of performing on-chip high-throughput computation. Additionally, many approaches compile the desired application into the design, which severely hinders flexibility, and limits them to extremely small states due to the FPGA resource usage scaling with quantum state size.

2.6. Conclusion

This chapter introduced the necessary technical and theoretical background for the remainder of this thesis, and provided an overview on the current status of relevant quantum computation simulation fields. Sections 2.1 to 2.2 provided the foundation for FPGA-based computing acceleration of DSP- and memory-heavy applications. Section 2.1 discussed DSPs and important memory types in the context of FPGAs were discussed (BRAM, UltraRAM, HBM). Section 2.2 explained the architecture and integration of AMD/Xilinx Alveo Data Center FPGA cards. Afterwards, Sections 2.3 to 2.4 established the required background on quantum computation simulation. Section 2.3 gave an introduction into concepts like state vectors, superposition, entanglement, quantum state operations, and non-ideal quantum systems. Additionally, an overview was provided on the different characteristics of Quantum Networking and Quantum Computing. Section 2.4 shifted the focus towards quantum computation simulation. It was pointed out that within the state vector formalism, there is a variety of computing methods, with several methods being specifically designed for certain simulation scenarios. An overview was given on common Quantum Network simulators, with a focus on the low-level, highly modular NetSquid simulator. Lastly, Section 2.5 assessed existing designs for simulating quantum computation on FPGAs - finding that most designs exhibit severe scaling drawbacks and lack flexibility, and that translating the software-based efforts towards efficient (simulation-dependent) state vector computation remains a task yet to be tackled.

3

High-Throughput Dense State Vector Computation

Section 2.4.2 pointed out that a multitude of computation schemes exists for quantum state vectors, with varying degrees of optimization. Based on these techniques, this chapter designs a computationally optimized, scalable quantum state vector computation scheme, that is specifically tailored for leveraging DDR-based acceleration platforms.

First, Section 3.1 conducts a study on relevant characteristics that a computation algorithm must fulfill, in order to efficiently utilize a hardware accelerator platform. Based on that, Section 3.3 introduces a novel, suitable representation for quantum state vectors. Using that representation, Sections 3.4 to 3.6 derive the necessary mathematical framework to perform quantum operations on the novel state vector representation. Lastly, Section 3.7 elaborates on the newly derived scheme's applicability to common types of hardware accelerators, along the criteria formulated in Section 3.1.

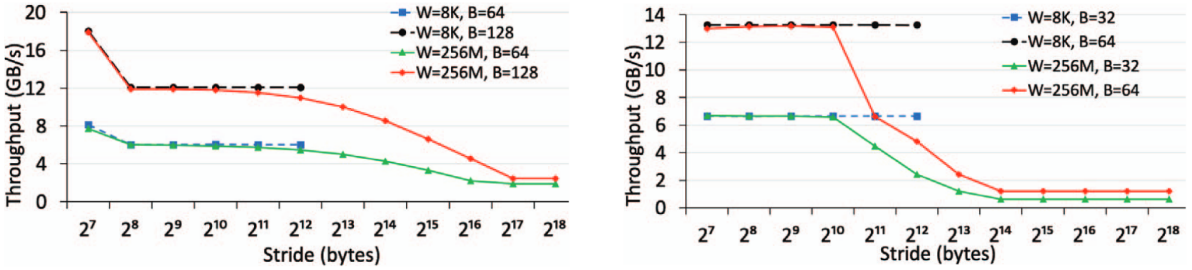
3.1. Algorithm Criteria

When specifically targeting hardware accelerator platforms, it is important that the algorithm - or the computation scheme, in this case - is tailored to optimally use the accelerator's computation and data throughput capabilities. This section derives criteria for such a scheme, with respect to handling quantum state vectors.

Optimizing computation capability utilization means that the algorithm should allow using as much of an accelerator's Compute Units (CUs) at the same time as possible (by pipelining, or in parallel). In terms of data throughput, the dataflow patterns of an application should be suitable to near the data source and sink's theoretical maximum datarates - which can be peripheral data in/out like ethernet (or PCIe), or on-chip/on-board memory.

For quantum operations, calculation time per operation is the dominating metric - although quantum operations sometimes can be "pipelined", to an extent (see [18]). Calculation time in turn is closely related to parallelizability, because operations are effectively matrix-vector multiplications, which require a large number of equal, relatively simple, operations. In contrast, multi-step calculations, like several cryptographic algorithms, would benefit at least as much from heavy pipelining, as they do from parallelization. True parallelizability requires maximum independence between the individual computations, because any dependence causes a need for synchronization between otherwise isolated CUs. From a computation perspective, a quantum operation simulation scheme for hardware accelerators must therefore allow performing computations in parallel, and eliminate data dependencies wherever possible.

In terms of maximum datarate, the accelerator's available memory is the predominant benchmark - rather than a certain peripheral datastream - because quantum states typically reside in accelerator memory. Common platforms, regardless of GPU or FPGA, offer a mix of DDR main memory (on-chip or on-board), and smaller local on-chip memory (Static RAM (SRAM) or similar technology). The desired quantum simulation scheme needs to be able to function well with both of these types of memory. For SRAM or similar memories, this does not pose a significant problem, as long as the limited total availability on a chip is taken into account. These memories have relatively constant access latency,



(a) DDR4 **(b)** HBM

Figure 3.1: from [29]; Effective memory throughput for DDR4 and HBM on an Alveo U280 device, with varying address strides. Individual accesses of size B bytes, confined to a memory region of total size W bytes.

with no or negligible variance across different memory access patters. That however does not hold for DDR memories, as is discussed in-depth in the following elaboration.

3.1.1. High-Throughput DDR operation

It is common knowledge that DDR-type memories operate at their highest bandwidth when being accessed sequentially. That is due to the hierarchical memory structure, with a row, or *page*, being the "granularity" at which the memory reads and writes data. Consecutive memory addresses either fall into the same page, or can be predicted, which allows the memory controller to pre-load the next memory page. This section quantifies the impact of these effects, in the context of quantum state vectors.

The study is based on a study by Wang et al. [29], using a modern AMD Alveo U280 FPGA accelerator card, that features both HBM and DRAM. Figure 3.1 from their publication shows the effects of access locality on both DDR4 and HBM throughput. They measure the effective throughput on individual accesses of size B bytes, confined to a memory region of total size W bytes, with varying address strides from one access to the next one. This data allows predicting the effective memory bandwidth when accessing quantum states of varying size in memory randomly, or sequentially. As a reference, this discussion assumes the common non-labeled full state vector representation with $128\,\mathrm{bit}$ vector elements, such that a q-qubit state takes up

$$2^q \cdot 128\,\mathrm{bit} = 2^{q+7}\,\mathrm{bit} = 2^{q+4}\,\mathrm{B}.$$

The first observation is trivial, when comparing the orange and black graphs in Figure 3.1, with the green and blue ones: HBM for instance has a read size of $64\,\mathrm{B}$ (8 bursts of columns), which is the access granularity for the orange and black graph in Figure 3.1b. The memory can not load chunks any smaller, thus, for an access of $32\,\mathrm{B}$, the memory still requires the time for loading the full $64\,\mathrm{B}$, while only half that amount of data is "effectively" read - hence the constant throughput drop by factor 2 for the blue and green graphs. The effect is the same for DDR, just with double the read size. The lesson to learn here is that, in any computation scheme, it is vital to prevent DRAM read and write accesses smaller than the respective memory's read size. As a side note, this is equally true for accelerator platforms as it is for non-heterogeneous systems, since both use DDR-type main memory.

The analysis of the stride size focuses on the HBM graph (Figure 3.1b). The effects are more pronounced here than in DDR (although conceptually the same), while an HBM stack (consisting of multiple channels) features higher maximum throughput than a DRAM. Since read size has been discussed, this paragraph focuses on only the orange and black graphs. For "large" quantum states, the black graph is irrelevant, because it concerns a memory region of only $8\,\mathrm{kB}$, or $2^{13}\,\mathrm{B}$. The graph does therefore not apply to any state that exceeds 9 qubits. For strides of $2^{11}\,\mathrm{B}$ and larger, the graph shows a significant performance drop, until it settles at about $10\,\%$ of the maximum throughput, for $2^{14}\,\mathrm{B}$ strides and larger. This is due to page miss accesses, which the memory controller fails to compensate for via prediction and pre-loading, until the point where every access is a page miss (hence the plateau). For array accesses, one can conclude that the effective throughput can drop by as much as $90\,\%$, if the average stride exceeds $2^{13}\,\mathrm{B}$. In order to formulate a correlation between array size and random-access stride length, the stride can be approached as the average distance between 2 array elements (assuming a uniformly distributed access pattern). The average distance of 2 elements in a range of size L is $L/3$ [30]. Therefore, to obtain the performance loss when randomly accessing an array of $2^n\,\mathrm{B}$, one gets a

good estimate by averaging the performance loss for stride lengths of 2^{n-1} and $2^{n-2}\,\mathrm{B}$. Translated to quantum states, random element accesses in HBM have no latency penalty up to 9-qubit states, and cost up to $90\,\%$ throughput for states of 12 or more qubits, compared to sequential accessing. For DDR4, according to Figure 3.1a, the throughput loss is more gradual and levels at roughly $14\,\%$ of maximum throughput for states of 15 or more qubits, but only states up to 3 qubits are not negatively affected.

When discussing realistic memory access penalties, cache effects can not be left out, since caching can significantly remedy memory latencies, and practically every conventional computing platform comprises some type of cache memory. Given that the limiting factor in using cache is the available cache size, one needs to put typical cache sizes in perspective to the quantum state sizes of interest. Two HPC platforms used in this thesis are the AMD Alveo U55C FPGA Data Center accelerator card, with around $41\,\mathrm{MB}$ of total local on-chip memory capability, and the Intel Xeon Gold 6230 multi-core CPU, with $27.5\,\mathrm{MB}$ of total cache. These figures are in the order of a 21-qubit state vector ($32\,\mathrm{MB}$). Realistically, one can expect that not all of the present buffering memory is actually available for quantum states. In the FPGA, BRAM and UltraRAM are distributed across the entire chip. Combining all of it into one logical memory would cause severe routing issues, and use up a lot of extra addressing logic. The CPU likely requires a portion of its cache for program control data structures, and possibly instructions. Still, even only 19 qubits in SRAM allows for larger states that do not suffer from random memory accessing, compared to a pure DRAM system.

The presented quantitative benefits of cache memory assumed a computation scenario with only one quantum state. The advantages diminish when the simulation consists of a multitude of states, which as mentioned in Section 2.3.3 is unlikely for a Quantum Computing simulation, but the default case in Quantum Networking. In a scenario of extremely dynamic state space behavior (including constantly removing and creating states), caching could theoretically even cause undesired overhead, because cache lines would constantly need to get invalidated and re-written. Nevertheless, within its size limits, the lower access latency and generally higher throughput of local SRAM-like memories are likely to hide DDR random access latency penalty - in many scenarios - and must therefore always be taken into consideration.

3.1.2. Conclusion

Concluding, a quantum computation simulator for large simulations on HPC accelerators requires a computation method that is highly parallelizable, relies on sequential state memory accesses, and ensures a memory access granularity no smaller than the minimum read size of the respective DDR. Up to a certain state size, random accesses are unproblematic, if an accessed state is small enough to fit into a DDR page, or into available buffering memory. For simulating state vectors of $128\,\mathrm{bit}$ complex-number entries, of roughly 21 qubits or more, random memory accesses can reduce the achievable data throughput by up to $90\,\%$. The same holds for simulations on smaller states, if the state space in memory occupies substantially more than the available buffer resources (for instance, $30\,\mathrm{MB}$ on Intel Xeon Gold 6230).

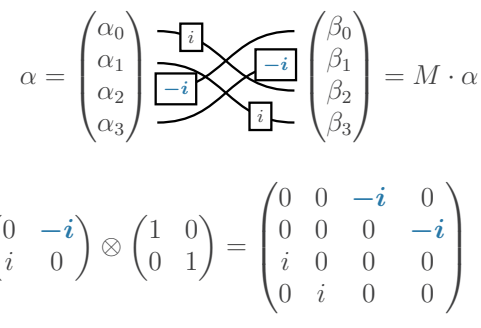
3.2. Existing Approaches

Section 2.4.2 has introduced a multitude of methods for simulating quantum state vectors. Referring to the previous discussion on efficient accelerator memory usage (Section 3.1.1), the presented methods have one common shortcoming: While all approaches offer good parallelizability, no technique allows pairing a high (linear) computation efficiency with fully sequential data accesses.

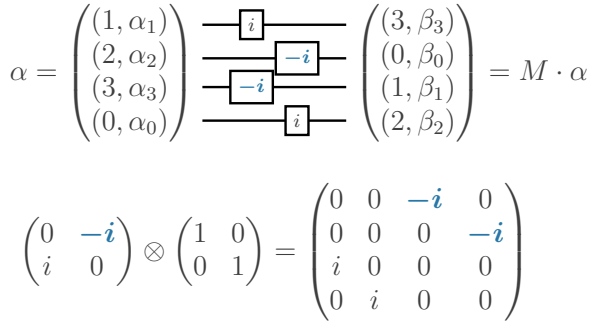
General matrix multiplications are computationally inefficient because the number of individual multiplications scales quadratically with the state vector size, as opposed to linear scaling with sparse matrix multiplication, and all quantum-specific approaches presented in Section 2.4.2. The impact on computation time is amplified by the fact that the vector size itself scales exponentially with the number of qubits (for all techniques except for sparse vectors).

Considering dataflow, on the other hand, only general matrix multiplication allows for fully sequential data accesses both for reading and writing, while none of the other techniques can fully avoid random memory accesses. Pairwise multiplication, both on full state vectors and labeled-hashed vectors, requires random access reading and writing. CSR requires random access reading. A column-based sparse matrix representation would cause random access writing.

In order to truly leverage the computation capabilities of accelerator platforms for simulating quantum



(a) *Ordered* vector representation - single-qubit gate as a weighted vector permutation



(b) *Labeled* representation `(label,amplitude)` - single-qubit gate as a parallelizable weighted label update

Figure 3.2: Dataflow analysis of a Y gate applied to qubit 1 of a 2-qubit state vector, in *conventional* and *labeled-random* representations. Dataflow paths are labeled with the multiplicative factor from the matrix $M = Y \otimes \mathbb{I}_2$.

state vectors, a scheme needs to be designed that has linear computation complexity, is fully parallelizable, and offers sequential data access patterns. It turns out that the existing quantum-specifc approaches already contain the basic ingredients for such a method: Firstly, all of them meet the desired computational efficiency. Jaques et al. [18] introduced the concept of a representation with “labeled” amplitudes, that enables parallelizability with non-ordered state vectors. They additionally highlighted that monomial gates (which they called *permutation gates*) are simpler to compute than general gates. Jones et al. [17] had previously pointed to the same characteristic. Jaques et al. defined permutation gates as functions, that perform label and amplitude updates on individual vector elements.

3.3. Labeled-Random State Vectors

Studying the concept of labeled-hashed vectors and “trivial” gate formulas leads to the following realization: When permutation gates are formulated as bijective mappings of individual labeled input vector elements, to individual labeled output vector elements, the input and output elements form disjoint pairs. Processing these pairs can therefore be done fully independently from one another, in arbitrary order, which allows to sequentially iterate through the state vector in memory, and process elements in parallel. These gates are in fact hindered by the hash function in labeled-hashed vectors, because the vector does not occupy a consecutive memory space (due to the hashing). This thesis proposes a labeled, unordered vector, which occupies consecutive memory, and is thus more desirable for these gates. That representation is henceforth called *labeled-random state vector*. Figure 3.2 illustrates the parallel dataflow and disjoint input-output element pairs in labeled-random state computation for the example of a Y gate, compared to a non-labeled representation.

Labeled-random representation can be used with both full and sparse state vectors. This work, targeting computing platforms optimized for high data throughput, uses full state vectors: It is the more generic option, it is suitable for simulating noisy systems, and modern accelerator platforms are expected to handle the large data structures well, as long as computations are parallelizable and memory accesses are sequential.

Implications

Removing the hash function comes at a cost: It eliminates the possibility to address specific vector elements by their label, with severe implications for non-permutation gates. Pairwise gate operation, which all quantum-specific simulation techniques in Section 2.4.2 utilize, is not possible without addressable vector elements. Because any output state element consists of more than one (generally two) summands, coming from multiple input elements, it is also not an option to express non-permutation gates as bijective projections between individual input and output state elements. At the time of writing, there is no algorithm for non-permutation gates in labeled-random representation that matches the efficiency of pairwise gate operation on labeled-hashed vectors. This work utilizes an implementation that offers efficient, sequential operation, up to a state size limited by the amount of available hardware, explained in Section 4.3.3. Another approach is presented in Section 9.1, which uses random access writing, is state size-independent, and has limited performance losses compared to labeled-hashed

states.

Additionally, labeled-random full state vector representation substantially increases the required quantum state memory (compared to non-labeled or sparse vectors). However, the cost in terms of quantum state size, while clearly noticeable, is less substantial than it might appear: Labeled representation adds one integer number per entry of a complex-valued vector. On most computing systems, that is a theoretical increase by less than a factor of 2, with a likely actual factor of exactly 2, when accounting for data alignment in memory. Full quantum state vectors double in size every time a qubit is added to the state. Consequently, labeled-random representation is unlikely to cost more than 1 qubit of maximum quantum state size that fits into a given memory. Nonetheless, due to the higher memory footprint, the raw amount of data movement is also doubled, compared to non-labeled vectors. Thanks to sequential data accesses, the "effective bandwidth" (vector data elements per second) with a DRAM still actually increases (for reasonably large states), as Section 3.1.1 showed. In summary, labeled-random state vectors, while costing up to one qubit in maximum state size, offer a significantly higher "effective" throughput, measured in vector data elements, which in many scenarios is a favorable tradeoff. For simulations that crucially depend on the maximum feasible state size, the option exists to use a simulator based on sparse state vectors, with the explicit aim to maximize supported quantum state size.

Alternative Solutions

An alternative to labeled-random state vectors would be a "constrained" hashed scheme, adapted for sequentially processed non-monomial gates. The hashing method would need to guarantee that a vector resides within a constrained memory range, reserved for only that vector - such that it would be possible to "ignore" the hash function during monomial gates, and simply iterate over the raw memory space. There are two problems to that approach: The unoccupied "gaps" in the memory range would need to be "marked" as such at all times, creating significant control overhead, and potentially additional memory accesses. Additionally, nothing is gained from reading sequentially, if elements are not written back sequentially as well. This however is mutually exclusive with applying the hash function while writing, meaning the result state is effectively not hashed, and must be treated as labeled-random. If at any point an operation requires element addressing, like non-monomial gates via pairwise multiplication, additional steps would be necessary like first "making the state addressable" again, by reading sequentially, and writing back with hashing, without doing any processing. This thesis does not contain any further research in the direction of constrained hashed schemes, but leaves this paragraph to indicate the possibility of such an approach.

3.4. Fundamentals on Element Update Formulas

During the conception of a hardware-efficient quantum simulation scheme in this thesis, the same idea arose from feasibility considerations, that Jaques et al. [18] used to "chain" or "fuse" consecutive quantum gates: Expressing a gate G as functions f and g, that update independent vector elements - thereby eliminating full operation matrices, and any dependencies between vector elements. In this thesis, the functions f and g will collectively be denoted as *element update formulas*. This section first motivates element update formulas, and then mathematically derives them for all quantum operations required in the system - per individual operation type.

Element Update Formula: *General Definition*

For applying an arbitrary quantum operation with base operation matrix M to qubit index r (0-indexed), in state vector α (in labeled representation), with the resulting state vector α^*, for any given element (k, v) in α, an *element update formula* yields the corresponding element(s) (k^*, v^*) in α^*:

$$k^*(k, r): \quad k \mapsto k^*$$
$$v^*(v, k, r, M): \quad v \mapsto v^*$$

(for specific operation types, suitable variations or additions to the above functions may apply)

Motivation

In the publication by Jaques et al. [18], element update formulas appeared as an “additional feature”, that allowed for chained quantum operations. From a parallelizability perspective, and therefore for this work, they are rather a necessity. Firstly, as was discussed in Section 3.1, dependencies of any sort would create non-sequential memory accesses, hence bottleneck the data throughput, which reduces “effective” parallelizability (a memory-bound scheme does not benefit from more CUs). Secondly, eliminating operation matrices is practically imperative for parallelized hardware. Operation matrices, in theory, scale in size with the state that an operation is applied to (quadratically for general multiplication, linear with sparse matrix techniques). Truly parallel CUs would each need their own “copy” of that matrix, which is infeasible. Therefore, at least groups of CUs, if not all, would need to share copies of the operation matrix, meaning very large, multi-accessed data structures, and hence a new bottleneck. Consequently, parallelization requires a local, low-footprint way to perform all processing steps for any vector element. Element update formulas provide that way.

3.4.1. Controlled Gates

The first part of discussing element update formulas consists in treating any kind of controlled gates, because control qubits are handled the same for any type of gate. They are therefore not mentioned further in the subsequent discussion.

In an amplitude label k (as a binary integer), the bits directly correspond to the qubit indices in the shared quantum state. The labels can algebraically be treated as separate states, with respective probability amplitudes. If a gate has one or multiple control qubits, it is only applied if all of these control qubits are in state $|1\rangle$. The immediate consequence is: If, for a set of control qubit indices $\{c_1, ...\}$, all bits c_i in a label k are 1, all the qubits c_i in that label’s elementary state are $|1\rangle$, and the gate is applied normally to the label and amplitude (as if it was a non-controlled gate). In any other case, label and amplitude simply remain the same.

Since this technique does not make any assumptions about the gate, like for instance the number of target qubits, it is universally applicable, with an arbitrary number of control qubits. It is additionally extremely trivial to perform in binary hardware. For this reason, in order to achieve universal quantum gate simulation, it suffices to derive computing schemes for arbitrary **single-qubit** gates, and for the swap gate (as it is the only gate with more than one target qubit, whose operation matrix can not be expressed as a controlled version of a single-qubit gate).

3.4.2. Corresponding Labels in Pairwise Gate Operation

During the early phase of this thesis, a set of algebraic element update formulas was formulated, solely based on observing isolated elements of the operation matrix. Later on, a significantly simpler formulation was discovered, which leverages pairwise gate operation (see Section 2.4.2). The newer formulation is described in this chapter. The prior formulation remains available in the appendix Chapter C, since it is still utilized in the hardware code at the time of delivering this thesis. Both formulations are equivalent.

As a preliminary step to formulating individual element update formulas, this section points out mathematical relationships and establishes thesis-specific algebraic terminology. These are necessary in deriving the formulas in Section 3.5 and Section 3.6.

In pairwise gate operation, the most relevant characteristic to facilitate the simpler element update formulas is the following theorem, which will be proven in this section:

Any single-qubit gate M, applied to qubit index r in an arbitrary state vector, can be simulated by applying M to pairs of vector elements with equal indices, except for bit r.

The formulation strengthens the concept of pairwise operation, by pointing out the special relationship between related indices/labels. For a vector row index (or label, in a labeled representation) k_1, to find index k_2 such that (k_1, k_2) form a pair that M can be applied to in isolation, one simply has to flip bit r in k_1. While writing this thesis, Xu et al. pointed to the same aspect [31] (also referring to Jones at al. [17]). Next to that article only being published after finishing this work’s implementation, this section adds a formal proof, because the approach is imperative for almost all subsequent element update formulas.

A good starting point to proving the above statement is to analyze the operation matrix composition. Figure 3.3 depicts a generic single-qubit quantum gate, applied to qubit 1 (0-indexed) in a 3-qubit state vector, and highlights several important “blocks” and “distances” to aid the following proof. All references to colors, “boxes” and “blocks” concern that figure, unless stated otherwise. The “non-overlapping” dark-blue blocks of identical submatrices along the main diagonal will be denoted *l-blocks*, because they

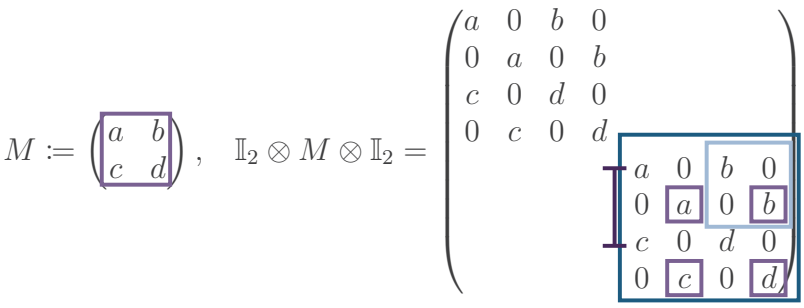


Figure 3.3: "Extension" of a dense base gate matrix M by identity matrices, leading to multiple "stretched out" occurrences of M in the result (highlighting boxes for $k = 7$, colors see text)

originate from a kronecker product with an identity matrix from the **left** side. "Non-overlapping" means that l-blocks span contiguous, disjoint ranges of columns and rows. *r-blocks* (in light-blue), caused by multiplying single elements m of M with an identity matrix as part of $M \otimes \mathbb{I}_n$, are submatrices to l-blocks.

It is useful to define expressions for the widths of l-blocks and r-blocks. The content of an l-block is equal to $M \otimes \mathbb{I}_n$, with $n = 2^r$ (r is the 0-indexed target qubit index, see definition above). It therefore has the width:

$$\textit{l-width} = \dim(M) \cdot 2^r \tag{3.1}$$

In this case, l-width $= 2^{r+1}$. r-blocks are in fact formed by computing $m \otimes \mathbb{I}_n$, with m any single element in M (again $n = 2^r$). A kronecker product that involves a scalar is a "normal" scalar product, with no effect to matrix dimensions, thus:

$$\textit{r-width} = \dim(\mathbb{I}_n) = 2^r \tag{3.2}$$

Additionally, two groups of elements in a matrix are defined as "coinciding", if any element in one group shares its column or row index with any element in the other group. Note that groups of size 1 are implied, which defines "coincidence" of single elements.

With those definitions made, the proof for the above statement can be tackled. A slighly more algebraic formulation of the statement is: In an operation matrix

$$M^* = \mathbb{I}_{2^l} \otimes M \otimes \mathbb{I}_{2^r}$$

for a single-qubit gate base matrix M, the elements at the intersections of any row and column indices (k_1, k_2) in M^*, where k_1 and k_2 differ exactly in bit position r (0-indexed, least to most significant bit), form again the matrix M (while all other elements in M^* are 0). The statement is proven in 2 steps:

1. **right side**

   It is valid to first prove the step from the initial case to the right side

   $$G := M \otimes \mathbb{I}_{2^r}$$

   , and then tackle the left side $\mathbb{I}_{2^l} \otimes G$, because the tensor product is associative. If the statement is true in one order, it automatically is in any order.

   (a) **distance**

   $M \otimes \mathbb{I}_{2^r}$ "replicates" elements m in M along diagonals 2^r times (forming r-blocks). For any 2 elements in M with either row or column indices (k_1, k_2), this creates multiple, non-coinciding instances with row/column indices $(k_1^{(n)}, k_2^{(n)})$, with the "increased stride" (dark-violet in Figure 3.3):

   $$k_2^{(n)} - k_1^{(n-1)} = 2^r \cdot (k_2 - k_1), \quad k_2 - k_1 \in \{0, 1\}$$

   Consequently, the "extended" matrix M (in light-violet) appears 2^r times in G, with a stride of 2^r between the different elements of one instance. Adding 2^r to a binary number is equivalent to simply setting bit r, if and only if the number has bit r unset. In other words, for any pair of non-equal row/column indices $(k_1^{(n)}, k_2^{(n)})$ in G ($k_1 < k_2$ without loss of generality), the initial claim holds if $k_1^{(n)}$ has bit r unset.

(b) **pairs in adjacent r-blocks**
By definition, two adjacent elements in M (row-wise or column-wise) must fall into adjacent r-blocks in G (because every element m "creates" the r-block $m \otimes \mathbb{I}_{2^r}$, and r-blocks are consecutive). Since r-blocks have the width 2^r (and start at row/column index 0), bit r is constant for row or column indices (respectively) within one r-block, and is different for adjacent r-blocks. Given that element $(0, 0)$ in M logically forms the r-block containing matrix element $(0, 0)$ in G, it follows that in any r-block in G, bit r of the column or row indices is equal to the respective (1-bit) index in M. Referring to the previous item, index $k_1^{(n)} < k_2^{(n)}$ implies $k_1 < k_2$, and therefore $k_1 = 0$ (0-indexed, because $\dim(M) = 2$). Accordingly, bit r of $k_1^{(n)}$ is unset, and thus, as shown, $k_1^{(n)}$ and $k_2^{(n)}$ only differ in bit r.

2. **left side**

For the left-side tensor identity product $\mathbb{I}_{2^l} \otimes G$, a small induction is used:

(a) **initial case** As already shown, the claim holds for G.
(b) **induction step** The product $\mathbb{I}_{2^l} \otimes G$ "replicates" G (dark-blue in Figure 3.3) along the main diagonal 2^l times, with row and column offsets $n \cdot \dim(G) = n \cdot 2^q$ for the n-th replication. Therefore, column and row index ranges of different l-blocks are equal modulo 2^q. Since $r < q$, it holds that, if M "appears" in G at pairs of row/column indices that only differ in bit position r, the same holds for $\mathbb{I}_{2^l} \otimes G$.

The above discussion encompassed **all** non-zero elements in M^*, because every operation in $\mathbb{I}_{2^l} \otimes M \otimes \mathbb{I}_{2^r}$ was fully considered. Consequently, the remaining elements in M^* are 0 as claimed, which concludes the proof.

3.5. Element Update Formulas - Gate Operations

This section derives element update formulas for "gate" operations. These include all types of quantum gates, but not quantum measurements.

3.5.1. Gate Operation Types

When applying standard quantum gates via pairwise gate operation, some operations turn out to be even more trivial than multiplying a 2×2 matrix with a 2-element (sub)vector. Section 3.2 already indicated that monomial gates are computationally simpler than non-monomial gates. Looking further, there are noteworthy sub-categories to monomial gates: The Z gate, for instance, only multiplies the individual vector elements with a respective factor, but does not change the "order" of the vector elements. In a similar way, the X gate simply swaps the two elements (and individually multiplies them with the respective scalar). Evidently, the simplicity in both cases originates from the diagonal, or antidiagonal, gate matrix, making the element update formula extremely simple. Based on this observation, the proposed system utilizes *gate operation types*, which categorize quantum gates by their matrix structure:

- monomial gates
  - diagonal gates (examples: Z, phase S or P, T)
  - antidiagonal gates (examples: X, Y)
  - swap gate
- dense gates (examples: H, general single-qubit rotations)

These categories cover all valid non-controlled quantum gates. Firstly, that is because dense gates are the generic case of a single-qubit quantum gate. Monomial gates are special cases, which happen to offer enormous potential for optimized simulation. The only non-controlled elementary quantum gate that does not have a single-qubit matrix is the swap gate, which hence forms its own gate operation type. Note that gates like CNOT/CX, CZ, and Toffoli/CCNOT, often treated like elementary gates, are all controlled versions of (other) elementary single-qubit gates. Paired with the universal method for treating controlled gates (Section 3.4.1), one gains a scheme to **efficiently simulate any quantum gate, applied to an arbitrary qubit index in a state of arbitrary size, without ever needing the extended operation matrix**.

3.5.2. Sparse Gates

Diagonal Gates

Diagonal gates M, as mentioned, only apply the factors m_{00} or m_{11} to the amplitudes labeled k_1 and k_2, without affecting the labels. Therefore, $k^* = k$. Matching the factors m_{00} and m_{11} to the respective labels/amplitudes is straightforward: Assuming $k_1 < k_2$, m_{00} is applied to k_1, and m_{11} is applied to k_2. It is guaranteed that k_1 and k_2 only differ in bit r (Section 3.4.2), with r being the target qubit index. For any given k, this makes it simple to determine whether k is the smaller or the larger label, in an element pair that M is applied to. If bit r is not set, k is the smaller label, and vice versa. As a conclusion, if $k[r] = 0$ ($k[r]$ denoting bit r in binary integer k, counting 0-indexed from LSB), the amplitude factor is m_{00}, and if $k[r] = 1$, the amplitude factor is m_{11}.

Element Update Formula: *diagonal* operation type

$$k^* = k \tag{3.3}$$

$$v^* : k[r] \begin{cases} 0 & \rightarrow v^* = v \cdot m_{00} \\ 1 & \rightarrow v^* = v \cdot m_{11} \end{cases} \tag{3.4}$$

Antidiagonal Gates

The argumentation is conceptually the same as for diagonal gates, only with different implications. The labels k_1 and k_2 are swapped, instead of kept. Since they only differ in bit r, this is achieved by flipping bit r in any k. The correct amplitude factor is again obtained based on bit r (before flipping). If r is set, k is the larger of the two elements in its pair, therefore column 1 in M applies, which leads to m_{01} - and accordingly, m_{10} if bit r is not set.

Element Update Formula: *antidiagonal* operation type

$$k^* = k \,\text{XOR}\, 2^r \qquad \text{“flip bit } r \text{ in } k\text{”} \tag{3.5}$$

$$v^* : k[r] \begin{cases} 0 & \rightarrow v^* = v \cdot m_{10} \\ 1 & \rightarrow v^* = v \cdot m_{01} \end{cases} \tag{3.6}$$

3.5.3. Swap Gates

The swap gate has two target qubits, r_1 and r_2. k^* for a given k is again obtained at bit level, but not via pairwise gate operation. Since the bit positions in k match target qubit indices, it is sufficient to simply swap bits r_1 and r_2 in k. Concerning the amplitudes, $v^* = v$ (Equation (3.8)), because all non-zero elements in a swap gate are exactly 1, and therefore do not alter any amplitude.

Element Update Formula: *swap gate* operation type

$$k^* = k \,\&\, \text{NOT}(2^{r_1} + 2^{r_2}) + k[r_1]\, 2^{r_2} + k[r_2]\, 2^{r_1} \tag{3.7}$$

“swap bits r_1 and r_2 in k”

$$v^* = v \tag{3.8}$$

3.5.4. Dense Gates

As indicated in Section 3.3, dense gates, or non-monomial gates, function not as straightforward as sparse gates. The element update formula for these gates has to assign two output labels and amplitudes $k^{*(i)}$ ($i \in \{1, 2\}$) to an input label k. Likewise, every $k^{*(i)}$ is assigned a separate matrix element $m^{(i)}$. Pairwise gate operation of course can still be leveraged to determine $k^{*(i)}$ and $m^{(i)}$. In order to compute an output element, two input elements are required, as Equation (3.11) below shows.

One way to develop the element update formula is treating dense gates as a “superposition” of a diagonal and an antidiagonal gate. Any label k produces both the results that it would in a diagonal, and in an antidiagonal gate. Therefore, one result label is k, and the other result label is k with bit r flipped. Because two results need to be accumulated, it is important to keep track of whether k is the label with bit r set, or bit r not set. As a convention, $k^{*(1)}$ is always the smaller label. Therefore, if bit r in k is unset, $k = k^{*(1)}$, and $k = k^{*(2)}$ otherwise.

Likewise, the amplitude factor corresponding with $k^{*(i)}$ are both the amplitude factor that would correspond to k in a diagonal gate, and in an antidiagonal gate.

Element Update Formula: *dense-gate* operation type

$$k^* : k[r] \begin{cases} 0 \rightarrow k^{*(1)} = k, & k^{*(2)} = k \oplus 2^r \\ 1 \rightarrow k^{*(1)} = k \oplus 2^r, & k^{*(2)} = k \end{cases} \tag{3.9}$$

with input vector α, result vector β

$$v_i^* : k[r] \begin{cases} 0 \rightarrow m^{*(1)}(k) = m_{00}, & m^{*(2)}(k) = m_{10} \\ 1 \rightarrow m^{*(1)}(k) = m_{01}, & m^{*(2)}(k) = m_{11} \end{cases} \tag{3.10}$$

$$v_i^* = \beta|_{k=k^{*(i)}} = m^{*(i)}(k_1) \cdot \alpha|_{k=k_1} + m^{*(i)}(k_2) \cdot \alpha|_{k=k_2} \tag{3.11}$$

How to embed the update formula, and deal with the element dependency, needs to be decided per implementation, because the most suitable way likely depends on factors like the computing platform.

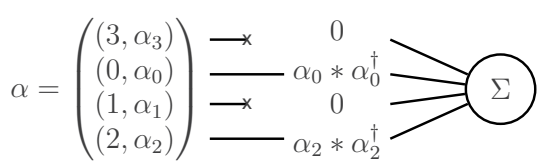
3.6. Element Update Formulas - Non-Gate Operations

Aside from quantum gates, there are two other relevant quantum state operations for a scheme to cover: Measurements, and merging quantum states (in accordance with several simulators henceforth called *tensor operation*). Measurements are a part of practically any type of quantum computing application. Tensor operations are not commonly seen in Quantum Computing simulations, since these normally establish one fully entangled state, shared by all involved qubits, during simulation setup. The simulation itself is carried out on that one entangled state, there are no two states to potentially merge. Quantum Networks in contrast, as explained in Section 2.3.3, often rely on the concept of individual nodes, which generate small entangled states (usually EPR pairs, thus 2-qubit states). Through techniques like entanglement swapping and distillation, the small states interact with other small states, meaning that it is common to see gates that have target or control qubits in different quantum states. In order to simulate a gate, all involved qubits must share one state, the two (or more) respective states therefore have to be joined at simulation time, which makes tensor operations an integral functionality to support Quantum Network simulation. The formulas derived in this section for measurement and tensor operations will also be referred to as element update formulas, although not all of these computations involve an operation matrix to be replaced.

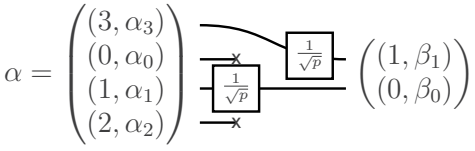
3.6.1. Measurement Operation

As indicated in Section 2.3.1, measurement operations consist of three steps: First, the probability p needs to be determined for a given measurement outcome (either 1 or 0). Afterwards, in simulation, a probabilistic step determines the measurement result (o in this section) - usually by comparing the measurement outcome to a value sampled from a continuous linear distribution between 0 and 1. Lastly, based on the measurement outcome, the state that was just measured needs to be translated into the matching post-measurement state, with respect to whether or not the measurement is destructive.

Regardless of destructive or non-destructive measurement, computing the post-measurement state is the step that most resembles applying a quantum gate, because it converts an existing state into a new state, with updates to vector element labels and amplitudes. The formulas for every step (except for the probabilistic, but otherwise trivial measurement outcome step) are given below in Equations (3.12) to (3.16), and are explained one by one in the following paragraphs.



(a) phase 1 - determine probability p_{pos} for measurement outcome 1/positive eigenstate



(b) phase 2 - compute the post-measurement state. p is the probability for the actual outcome, so $p = 1 - p_{\text{pos}}$ in this example. for non-destructive measurement, the output state β would have 4 elements, with the additional elements being 0

Figure 3.4: 2 phases of vector operation during *measurement* operation, in labeled notation. example for destructive measurement of the "least significant" qubit on a 2-qubit state, with outcome 0/negative eigenstate (only the vector operations are displayed, determining the measurement outcome is omitted, which happens in between)

Measurement Probability

As Equation (3.12) below shows, there is a relatively simple way to determine the measurement probability, by simply looking at one bit in each vector elements' indices. Depending on the value of that bit, the respective element amplitude is either included or not included in the computation of p. An algebraic analysis shows why that is:

The general case is measuring a qubit in a mult-qubit state. In that case, according to Equation (2.8), the projector - denoted F in this section - needs to be "extended". In the previously used example for targeting qubit 1 (0-indexed) in a 3-qubit quantum state:

$$
\begin{aligned}
F\big|_{f=1} &= \mathbb{I}_2 \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \mathbb{I}_2 \\
&= \begin{pmatrix}
1 & 0 & & & & & & \\
0 & 1 & & & & & & \\
 & & 0 & 0 & & & & \\
 & & 0 & 0 & & & & \\
 & & & & 1 & 0 & & \\
 & & & & 0 & 1 & & \\
 & & & & & & 0 & 0 \\
 & & & & & & 0 & 0
\end{pmatrix}
\end{aligned}
$$

This matrix will be denoted as $F^{*(1)}$ in the following ($F^{*(0)}$ would be the analog matrix for $f = 0$). $F^{*(1)}$ consists of only zeros, except for a number of "1" entries along the main diagonal. Those entries exactly appear in the rows whose indices, as a binary integer, have the target qubit index bit set to 0 (the reason is part of the proof in Section 3.4.2). Applying $F = F^{*(1)}$ to $\langle\psi| F |\psi\rangle$ yields:

$$
\begin{aligned}
& \langle\psi| F^{*(1)} |\psi\rangle \\
&= \begin{pmatrix} \psi_0^\dagger & \psi_1^\dagger & 0 & 0 & \psi_4^\dagger & \psi_5^\dagger & 0 & 0 \end{pmatrix} \cdot |\psi\rangle \\
&= \sum_{i \in \{0,1,4,5\}} \psi_i^\dagger \psi_i
\end{aligned}
$$

In Equation (3.12), $\sum_i (1 - k_i[q])$ replaces $\sum_{i \in \{0,1,4,5\}}$. Remember that $\langle\psi|$ is the conjugate transpose of $|\psi\rangle$. Figure 3.4a shows how the newly derived formula results in a dataflow that allows for high parallelization and sequential data accessing, although a subsequent accumulation step can not be avoided.

Post-Measurement State - Non-Destructive

A non-destructive measurement keeps the measured qubit in the state. There is hence no change in the labels, so $k^* = k$ (Equation (3.13)). Concerning the amplitudes, any post-measurement state calculation requires a normalization step with $1/\sqrt{p_o}$ (both for non-destructive, and for destructive measurement). For non-destructive measurement, the normalization however only applies to vector elements that correspond to the state that qubit q has been measured to (which is exactly half of the state vector elements). The other half is set to 0, because during the measurement operation, qubit q has resolved

to the opposite partial state. A state vector element corresponds to measurement outcome $o = 1$ exactly if $k[q] = 0$, or "o XOR $k[q]$" in digital logic.

Post-Measurement State - Destructive

Computing the result state of a destructive measurement is trivial for vector element indices that do not correspond to o: These input state elements are simply dropped, no result state element is created for any of these. This is what automatically shrinks the state by one qubit, when computing the post-measurement state (illustrated in Figure 3.4b). How to determine whether or not a label k corresponds to o is described in the previous paragraph.

For indices that correspond to o, computing k^* again has a reasonably simple expression at bit level: "From the binary integer label, eliminate bit q, by shifting down the more significant bits by one bit position". Equation (3.15) accomplishes this operation mathematically as follows: $(k \bmod 2^q)$ are the bits less significant than q, which remain unchanged. $\left\lfloor \frac{k}{2^{q+1}} \right\rfloor$ "extracts" the bits of higher significance than q, and shifts them down to the 0th bit position. Multiplication with 2^q shifts those bits up to bit position q, which in combination has set any bit index $< q$ to 0, and shifted any bit index $> q$ down by one bit. The two terms added up yield the desired binary operation.

Computing v^* in contrast is straightforward. Correspondence with the measurement result is handled via the element label update, which leaves only the normalization factor to apply (Equation (3.16)).

Element Update Formula: *measurement* operation type

q: target qubit index $\quad o$: measurement result

$k[q] \coloneqq \left\lfloor \frac{k}{2^q} \right\rfloor \bmod 2$ "binary integer k at 0-indexed bit position q"

Measurement Probability

$$p(o = 1) = \sum_i (1 - k_i[q]) \cdot v_i v_i^\dagger \tag{3.12}$$

Post-Measurement State - Non-Destructive

$$k^* = k \tag{3.13}$$

$$v^* = (o \,\text{XOR}\, k[q]) \cdot \frac{1}{\sqrt{p_o}} \cdot v \tag{3.14}$$

Post-Measurement State - Destructive

for $(o \,\text{XOR}\, k[q]) = 1$

$$k^* = k \bmod 2^q + 2^q \cdot \left\lfloor \frac{k}{2^{q+1}} \right\rfloor \tag{3.15}$$

$$v^* = \frac{1}{\sqrt{p_{\text{meas}}}} \cdot v \tag{3.16}$$

3.6.2. Tensor Operation

The last operation to discuss is the tensor operation, which joins two quantum states into one. With the states denoted as α and γ, Figure 3.5 demonstrates that each element in α has one result element **per** element in γ. Just like with sparse quantum gates, every result vector element only depends on one multiplication, and one input vector element (per input state). As a consequence, the computation can be expressed as multiple sets of multiplications between the vector α and a scalar (the individual elements of γ), alongside with a label update.

- **Element Label**

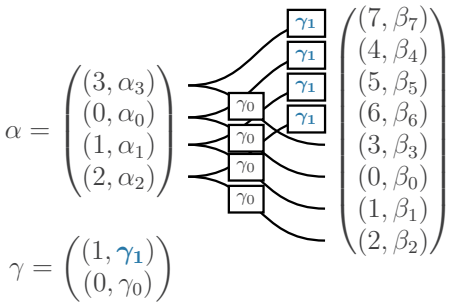


Figure 3.5: dataflow of *tensor* operation in labeled state vector representation. States α and γ are merged into the joint state $\beta = \gamma \otimes \alpha$.

The individual scalar multiplications produce 1-dimensional subvectors, similar to what earlier in this thesis has been introduced as an l-block. As part of the similarities, the subvectors are contiguous, non-overlapping structures. Any subvector therefore "fully covers and fills" a consecutive range of result vector elements - every element in the block lies in that range, and every element in that range is defined by only this block. Because a range is a scalar vector product, its size is exactly the size of the vector, so a range has $\dim(\alpha)$ elements. Since the ranges begin at element 0, they are furthermore aligned to $\dim(\alpha)$.

The implication for the computation is: The update to the label k_α only consists of applying the current range "offset", which is given by the range size, multiplied by the label k_γ. Written as one term (Equation (3.17)):

$k^* = k_\alpha + \dim(\alpha) \cdot k_\gamma$

- **Element Factor**

  Compared to the label update, the amplitude update is rather simple. It is achieved by multiplying the amplitudes of the elements at hand from vectors α and γ (Equation (3.18)):

  $v^* = v_\alpha \cdot v_\gamma$

Element Update Formula: *tensor* operation type

$$k^* = k_\alpha + \dim(\alpha) \cdot k_\gamma \tag{3.17}$$
$$v^* = v_\alpha \cdot v_\gamma \tag{3.18}$$

3.7. Computational Efficiency

The derived element update formulas only contribute to a hardware-efficient computing scheme, if they are feasible to be performed per parallel compute unit locally, and fast. In order to be fast, they have to be simple, from the perspective of digital hardware. In order to support local execution, they need to have a "small footprint" - which can mean few instructions, hardware usage, and small required data structures, depending on the platform type. All of these requirements are fulfilled for conventional accelerator systems, regardless of whether it is a multi-core CPU, a GPU, an FPGA, or maybe even an Application-Specific Integrated Circuit (ASIC):

- **Simplicity**

  The formulas are assembled from trivial bitwise operations like shifts and single-bit logic gates, from conditional statements, and from decimal number multiplication. Support for the first two is self-evident. GPU CUs are built to perform floating-point multiplications, multi-core CPUs typically also contain floating-point multipliers. FPGAs have specific DSPs for integer multiplication, which in binary is the same as fixed-point multiplication. For ASICs, there are numerous circuits for both decimal number representation types. The inverse square root step for post-measurement states

	general	sparse matrix	labeled-hashed	labeled-random
computation complexity	quadratic	linear	linear	linear
parallelizability	(yes)	yes	(yes)	yes
matrix buffer complexity	quadratic	linear	constant	constant
vector memory complexity	$\times 1$	$\times 1$	below $\times 1$	$\times 2$
data access	sequential/ interleaved	depends/ interleaved	random access	(sequential)

Table 3.1: comparison of computation characteristics for quantum operations on state vectors using general matrix-vector multiplication, sparse matrix-vector multiplication, and the proposed labeled-random quantum state vector computation with element update formulas.
(explanation below instead of in the text, because the information stems form multiple sections)
general: requires a non-parallel (blocking) final accumulation step, hence only "(yes)".
general and *sparse matrix* data access generally require interleaving between state vector and operation matrix.
sparse matrix: data access pattern depends on the specific representation.
labeled-hashed: parallelizability is bottlenecked by random memory accesses. vector memory complexity "below $\times 1$" for sparse vectors as published.
labeled-random: data access is implementation-dependent for dense gates, but "sequential" for any other operation. vector memory requires an additional integer per complex entry, hence the broad "$\times 2$" for full states

is practically never efficient, because it involves computing the square root of an integer number in either representation, but it can not be avoided.

- **Low Footprint** On all considered platforms, the formulas only take up few local resources:
  - On a GPU, the SIMD kernel (the element update formula) consists of few instructions, and the CUs only need local copies of the 4-element base gate matrix. The situation is similar for a CPU, where the local gate matrix copies would rather reside in a high priority cache, due to the frequent accesses.
  - On an FPGA or an ASIC, the 4-element base matrix can either be local to every CU, or a read register shared by groups of CUs. The hardware cost of the element update formula circuitry is marginal (except for the multiplication itself), because they all rely on operations targeting single bits.

  The exception is again the inverse square-root step, which fortunately is a one-time, central step, and is hence not subject to the low-footprint requirement.

3.8. Conclusion

This chapter demonstrated that the introduced *labeled-random* state vector computation, including hardware-efficient element update formulas, can significantly help to exploit modern accelerator platforms for universal gate-level quantum simulation. Table 3.1 summarizes the characteristics and improvements, compared to general matrix multiplication, sparse matrix multiplication, and labeled-hashed (sparse) state vectors. At the cost of a higher memory footprint, the newly developed scheme offers the novel combination of state-of-the-art computation complexity, great parallelizability, **and** DDR/DRAM-friendly data access patterns.

4

Accelerator High-Level Design

The previous chapter introduced *labeled-random state vector* computation for quantum simulation on HPC hardware. In this chapter, a high-level accelerator system is designed to implement that scheme, and integrate it with a front-end application to run on a host CPU. Given the multitude of common accelerator platforms, Section 4.1 is dedicated to identifying the most suitable choice for a prototype system. The remainder of the chapter lays out the proposed design. Conceptually, this chapter keeps the discussion at system level, and as generic as possible, with respect to the chosen platform type. Chapter 5 hereafter in turn is dedicated to implementation level, and elaborates on design components in detail.

4.1. Platform Type

Since labeled-random state vector operation is a computation scheme, not an implementation, it is not tied to one specific type of accelerator platform - while being designed for platforms which rely on DDR main memory. That applies to all of multi-core CPUs, GPUs, and FPGAs. Between those three, this section aims to identify the most promising one for a prototype implementation. In a first step, Section 4.1.1 formulates platform requirements for efficient, large-scale labeled-random state vector computation. Afterwards, Section 4.1.2 analyzes, per requirement, how well it is fulfilled by the three named platforms. Finally, a decision is made, based on the analysis.

4.1.1. Requirements

In order to decide on a platform type to use in this thesis, the following criteria were set up, with not only the pure computation performance in mind. Rather, the study encompassed which platform would best facilitate future optimization and design space exploration, as well as offer the most flexibility for varying simulation scenarios.

- **Hard Requirements**
  - full operation support for element update formulas
  - high-throughput SIMD computing capabilities (with synchronization support for accumulation)
  - large, high-throughput memory resources
  - good host system integration (to couple the accelerator with a front-end simulator application)
- **Soft Requirements**
  - low-latency memory accesses (considering any available memory)
  - multi-threading (for potentially performing multiple operations at the same time, on different states)
  - non-blocking execution (because gate operations have no immediate “result” - a host application only needs post-operation data from the accelerator upon a measurement, or when reading out the state)
  - networking/clustering support (to couple multiple hardware instances)

4.1.2. Decision

Requirement Analysis

The following elaboration lists, per requirement, to which extent it is met by the three aforementioned platform types:

- **Hard Requirements**
  - operation support: As explained in Section 3.7, all platforms support the necessary operations, while also all of them lack a native way to perform the inverse square root step.
  - high-throughput SIMD: Multi-core CPUs, evidently, offer SIMD via a limited number of parallel cores, while GPUs are entirely built around extensive SIMD execution. FPGAs have good support for parallel computation circuits, aided by their columnar internal structure and hard-IP DSP elements, but can be outperformed by the raw computing power of a GPU.
  - high bandwidth memory: All platforms support high-speed DDR type memories. In terms of bandwidth, a multi-core CPU system can be bottlenecked by the PCI bridge between CPU and memory. GPUs and HPC FPGAs however have exclusive, closely integrated DDR memory (on-board or on-chip), which allows for a higher usable bandwidth. Additionally, a CPU-based system is unlikely to employ HBM memory, because the achievable memory bandwidth exceeds all existing PCIe standards ($450\,\mathrm{GB/s}$ dual-stack HBM on AMD Alveo U55C, PCIe 7.0 x16 is projected to support $242\,\mathrm{GB/s}$). Therefore, GPUs and FPGAs have a higher data throughput ceiling.
  - host system integration: all platforms are designed with host system integration in mind, while some are simpler to work with than others. Multi-core CPU code is often handled by simple preprocessor directives (OpenMP, for instance). Programming GPUs comes at some extra difficulty, because it generally requires specifically implementing compute kernels. Widely established frameworks make that process accessible, like OpenGL [32], and Nvidia's CUDA [33]. FPGA accelerators, due to their flexibility, do not have that kind of universal language. Still, AMD/Xilinx for instance (being the leading vendor of Data Center FPGA hardware) provides the Xilinx Runtime Library (XRT), a framework based on OpenCL, for interaction between Alveo accelerator cards and a host system. With the still significantly higher complexity however, compared to CPU and GPU programming, comes a great increase in flexibility.
- **Soft Requirements**
  - low-latency memory: Considering local buffering memory, all platforms typically offer a solution of some kind. CPUs and GPUs have (multi-level) SRAM cache memories. FPGAs have BRAM and UltraRAM resources, which need to be manually embedded into a design. For DDR memories, the close integration is expected to give FPGAs and GPUs an advantage over multi-core CPU systems. The GPU combines the best of both worlds, given that it has a comparable DDR integration level to an FPGA, combined with the cache memory convenience of a CPU.
  - multi-threading: All platforms offer their own flavor of multi-threading. A CPU allows spawning multiple actual program threads, albeit with some overhead. For GPUs, CUDA, as an example, supports loading multiple kernels at the same time. FPGAs can be programmed to use multiple computation circuits in parallel, which causes some implementation overhead for data multiplexing, and in the accelerator-host interface. On the positive side, the overhead at execution time is minimal, compared to the other two candidates.
  - non-blocking execution: GPUs with CUDA support non-blocking operations (leaving the host free carry out other simulation-related tasks, while the accelerator works in the background). FPGAs offer the freedom of implementing arbitrary execution models, therefore non-blocking execution is a possibility as well. A multi-core CPU suffers from the fact that the host code, which would mostly be non-parallelizable execution control, and the non-blocking computation thread, run on the same system. There is no external hardware to "offload" computations to. Therefore, non-blocking execution is possible in theory via multi-threading, but comes with the overhead of multi-thread programming, and causes effects like disadvantageous cache interaction between host code and computation.

  - networking/clustering: CPUs offer linking with other devices via standard network stack. GPUs and HPC FPGAs both offer sophisticated peer-to-peer data exchange methods, which comes at the benefit of not straining the interface between accelerator and host (PCIe, usually). Nvidia CUDA GPUs can be linked via NVLINK hardware, either in pairs via bridges, or in larger groups via dedicated switching hardware. Data Center FPGAs like the AMD Alveo line have dedicated built-in gigabit serial transceivers (additional to the PCIe hardware) with on-board SFP/QSFP serial interface connectors, allowing to link multiple devices in a ring topology, or via switching hardware.

Choice

As the analysis shows, a GPU and an FPGA both appear to be suitable candidates, while a multi-core CPU exposes some non-ideal characteristics in multiple areas. Firstly, the achievable memory bandwidth, and therefore the maximum theoretical computing performance, is expected to be lower with a CPU system, mainly for architectural reasons. Secondly, the amount of parallel CUs is less than what GPUs and FPGAs offer, while the parallelizable application - the element update formulas - is “simple” enough to favor many less sophisticated CUs (like on a GPU) over fewer more complex ones (like CPU cores). Lastly, especially Quantum Network simulations, which have to maintain multiple states and agents, can significantly benefit from the ability to split quantum state simulation from non-parallelizable simulation overhead. It is valid to conclude that the application lends itself “well enough” to a dedicated accelerator platform to justify spending the extra hardware and implementation time, over a pure CPU system.

Between a GPU and an FPGA, there is a realistic chance that, in the right simulation scenario, the GPU would outperform the FPGA in pure computation time. That is especially true for “homogeneous” simulation scenarios - like a quantum computing simulation on only one large state - because that allows the GPU to utilize a maximum of its CUs, with a minimum of control overhead, and with efficient cache usage. Additionally, while writing a GPU program is not trivial per se, the implementation process usually still takes a fraction of the time of an equivalent FPGA implementation. Nevertheless, an FPGA was chosen as the platform for the prototype that is presented in the following chapters. The reasons lay in supporting a wider range of simulation scenarios, and greater future extension possibilities:

- **Memory Structure**

  The FPGA offers an almost immediate access to on-board DRAM or HBM. While less impactful in memories with one single access channel, it can play a crucial role in exploiting a multi-channel memory - like HBM. One can manipulate quantum state allocation in memory to separate reading and writing channels from each other, and to always use as many channels as possible for maximizing bandwidth utilization. Secondly, local on-chip memory can directly be used for small states (see Section 7.2.4), instead of acting as DRAM cache - reducing both control overhead (at the cost of implementation difficulty), and DRAM fragmentation.

- **Extensibility/Flexibility**

  Quantum algorithms, especially in Quantum Networking, consist of more than only the quantum gate operations. Many common algorithms, down to a trivial entanglement swap, include (potentially) non-quantum actions that depend on measurement outcomes. NetSquid, as an example, employs an entirely event-based simulation model, with multiple independent agents. These portions are still executed by the host CPU. The problem is not necessarily the CPU time this takes, but rather the communication overhead it causes between host and accelerator (more in terms of how often both need to communicate, than total amount of exchanged data). An FPGA can easily dedicate a portion of hardware to taking over some of these tasks, for instance by maintaining an instruction queue, by implementing a “quantum simulation processor” with a specific reduced Instruction Set Architecture (ISA), or by simulating entire agents on a soft core. Likely, none of these affect the quantum state operations, except for the general availability of hardware resources. A GPU does not have the flexibility for this kind of “integrated side-by-side” implementations. As outlined, some simulation scenarios could benefit from this type of integrated simulator much more than others.

	parameters	operation
`measure`	target	measure qubit
`operate`	matrix, target(s), control(s)	apply gate
`tensor`	target states	merge states
`clear`	-	clear system (reset)
`create`	size	create new state (initialize to $\|0...\rangle$)
`readout`	index	
`set`	index, data	write to existing state

Table 4.1: core layer operation set - the front-end interface receives the instructions from low level software, and passes them on to the core layer via core control

4.2. Design Overview

As elaborated in Section 4.1, an FPGA was chosen as the target platform for the accelerator system, which is presented in this and the following sections. However, Section 4.1 discussed that FPGAs and GPUs were both promising options, since they are both capable of highly parallelized, high-bandwidth computation. For this reason, while this and the following sections explicitly assume an FPGA platform, substantial parts of the design are laid out in a platform-agnostic way - which facilitates a potential future GPU implementation. In any case, this section is written with “custom digital logic” in mind. After introducing the proposed accelerator, Section 4.5 provides a brief discussion on the design’s applicability to GPUs.

A high-level overview of the proposed design is given in Figure 4.1. It is divided into three layers: *front end* (software), *intermediary layer*, and *core* (both hardware). The core layer is the computational center of the design. It contains the *engine*, which holds implementations of all gate and non-gate element update formulas (Sections 3.5 to 3.6), and hence carries out any operation on quantum states in memory. The *quantum state memory* unit is responsible for on-board quantum state data. That does not only entail managing on-chip/on-board memory allocation and operation, but also handling data traffic both to and from the engine, and between core and intermediary layer. The intermediary layer, from low to high level, consists of the *core control* unit, and the *front-end interface*. The front-end interface is responsible for any type of communication with the (software) front end - both control and data. It activates core control for any operation, and supplies it with operation parameters. Core control orchestrates core operations (including non-quantum operations like `clear` and `readout`), by activating and monitoring engine and quantum state memory, and reporting the cumulated operation status, or any operation results, to the front-end interface. It is furthermore responsible for distributing operation parameters to engine and quantum state memory.

On the software side, a *low level software* layer (*drivers*) interacts with the hardware front-end interface on one side, and provides an application-friendly API to the *user application*. The user application in turn is “whichever quantum simulator the accelerator gets integrated into, as the quantum simulation back end”, and is therefore interchangeable. The (specific) user application is not part of the accelerator system design itself.

The remainder of this chapter delves into the design of the accelerator’s “functional” components (the core layer), namely engine and quantum state memory. Their high level design is not implementation-specific, and does not depend on a specific platform/FPGA instance. Low level software and front-end interface are dictated by the exact hardware, since (FPGA) platforms vary in the way they integrate with a host computing system. Core control operates on implementation level, not on system level. The inter-module protocols that it coordinates are part of the implementation design, and any challenges in distributing operation parameters would stem from the specific hardware platform.

4.2.1. Abstract Core Layer Interface

The core layer has a clearly defined operation set, that eventually is exposed to the software (Table 4.1). Establishing this operation set early on gives a clear structure to engine and quantum state memory, and helps to pave the way for the intermediary layer, whose entire purpose is to enable the “communication”

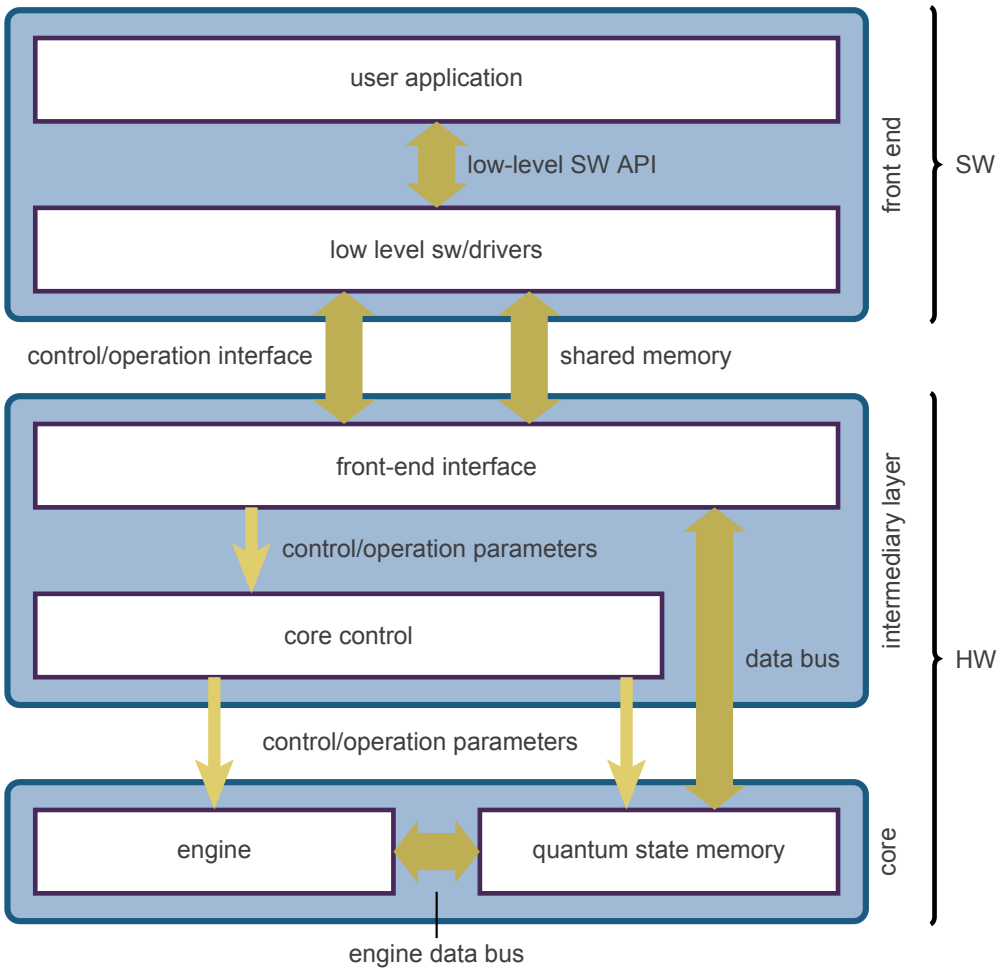


Figure 4.1: Design overview. The system consists of 3 layers, out of which 2 reside in hardware (*intermediary layer*, *core*) and one in software (*front end*).

between core layer and software.

Most of the operations are self-explanatory. Yet the decision to have a separate `create`, next to `set`, might require additional explanation. One reason is that a simulator front end might do so as well (NetSquid for instance does), in which case this design helps with connecting front end and accelerator. Secondly, there are scenarios in Quantum Network simulations where this design allows shifting workload from software to the accelerator. In Quantum Networking, states are regularly created as part of the simulation (in contrast to Quantum Computing), potentially with a probabilistic initial state. Without a separate `create` operation, software **has** to use a probabilistic algorithm, determine the generated state, and transfer the state to hardware. With a separate `create` operation (and an additional future parameter), the accelerator can use an internal entropy source (with the possibility to reuse the one it already has for measurement outcomes), and generate the matching state itself - sparing software of the probabilistic algorithm, and saving the time for the state vector transfer.

4.3. Computation Engine

This section discusses the core layer's first functional model, the *engine* (Figure 4.1). It is comprised of the element update formulas, implemented as hardware functionalities. Since the formulas are designed for parallel operation, they are realized as multiplexed arrays - one array per operation type in Section 3.5 and Section 3.6 - of small, parallel CUs, which operate on one wide (full-duplex or dual-simplex) data stream. The following paragraphs depict the respective high-level hardware structures.

From an architectural perspective, it is important to realize that there is no "engine control" module (in contrast to the quantum state memory unit later in the chapter, see Figure 4.7). Partially, that is because the "control" aspect is already encoded in multiplexing between functional CU arrays (the multiplexing is not displayed explicitly). Furthermore, implementations are encouraged to handle fine-grained engine control via control signals in the data bus, that links to quantum state memory, because regardless of other circumstances, that is a straightforward way to automatically keep the engine's data signals and control signals in sync.

Unlike the introduction of element update formulas in Sections 3.5 to 3.6, the computation arrays are discussed below in the order of increasing hardware complexity.

4.3.1. Sparse-Gate Operations

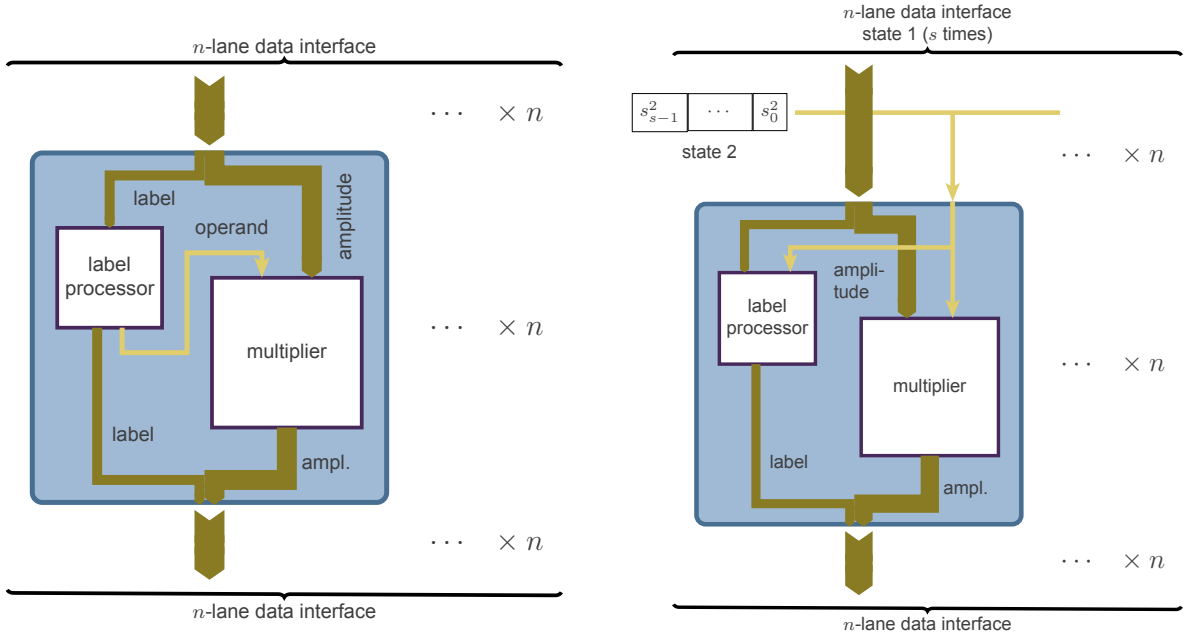
Sparse gates are relatively straightforward to process, as their element update formulas suggest. Individual vector elements are streamed through parallel "kernels", in which a "label processor" determines both the new element label, and the multiplicative amplitude factor from the base gate operation matrix, and a "multiplier" computes the new amplitude (Figure 4.2a).

4.3.2. Tensor Operations

Tensor operations are distinctively different from all other operations, in that they require two input states. Nonetheless, the design relies on only one input databus, such that interleaving between both states is required (Figure 4.2b). In the proposed scheme, one state, denoted as *state 1*, is streamed, analogous to the input state in a sparse gate operation. The other state (*state 2*), is regarded static, while state 1 is being streamed. One element of state 2 is forwarded to every parallel Processing Element (PE), and kept constant while state 1 is streamed. Logically, state 1 needs to be streamed multiple times, iterating through all elements in state 2, in order to carry out the full computation. Concerning state 2, it is loaded into a local buffer in batches, in an attempt to avoid reading single elements of state 2 from main memory in between of state 1 iterations, which would cause inefficient memory accessing.

4.3.3. Dense-Gate Operations

As explained in Section 3.3, computing dense gates in labeled-random state representation requires a way to deal with the fact that one can not address any particular input vector element. That makes the module significantly more complex than the hitherto presented operations. For this implementation, two approaches were considered for simulating dense gates: Firstly, the method presented in this section, which stems from systolic multiplier arrays, and secondly, a random access writing-based scheme (explained in Section 9.1) that is structurally similar to pairwise gate operation in labeled-hashed representation (see Section 2.4.2). The array approach has the benefit of fully sequential memory accesses, but the drawback of quadratic (instead of linear) computation complexity for state sizes beyond



(a) *sparse-gate* (engine operation types *diagonal*, *reverse-diagonal*, and *swap*)

(b) *tensor state 2* displayed as simple arrow, in contrast to *state 1*, because *state 2* is static per processing iteration, not streamed like *state 1*

Figure 4.2: High Level schemes for embedding *sparse-gate* (Section 3.5.2) and *tensor* (Section 3.6.2) element update formulas in parallel, scalable computing structures. The label processors/checkers implement the element update formulas, and have access to any relevant operation parameters (such as target qubit, operation matrix)

the feasible array size - because it essentially performs a non-optimized general matrix multiplication. The random memory access method has linear complexity, but is slower than the array method for “manageable”-sized states, and also theoretically slower than random access pairwise gate operation in labeled-hashed representation. Eventually, the motivation to choose the array approach for this prototype implementation lay in strictly adhering to sequential memory accesses, to allow exploring the resulting possibilities.

Design

The concept is based on a 2-dimensional array of CUs, instead of the 1-dimensional arrays for sparse gates and tensor operations. The individual elements are as illustrated in Figure 4.3. Referring to that figure, the array has the same width n as the sparse gate array. The depth M is parameterizable, but expected to be a power of 2. Similar to the tensor operation, the input state vector is streamed through the array multiple times (unless its number of elements s does not exceed M). During each iteration i, each “array level” m (the set of CUs across the width n) is assigned exactly one output element - namely the element with label $k^* = i \cdot M + m$ (“label generator” in the figure). This way, the computation “iterates” through the result vector elements, in contiguous batches of size M. In this scheme, the label analyzers have the task of “filtering” the input labels for the two labels $k^{(1)}$ and $k^{(2)}$ that are required to compute the element k^*. Upon encountering one of those labels, the label analyzer forwards the matching operand from the base gate matrix to the multiplier, which multiplies it with the element amplitude, and sends the result to a level-wide accumulation. In any other case, the multiplier is “deactivated”, meaning that its result is 0. The incoming vector elements are cascaded through the array levels, like in a systolic array. After every iteration of the input vector, the M array level accumulation results are written to the output data stream, and the accumulations are reset.

Looking for corresponding input labels for a given output label means “inverting” the element update formula (Equations (3.9) to (3.10)). Conveniently, the formula already is its own inverse for labels, and only requires minimal adaptation for amplitude factors: Any k^* is obtained by keeping one label $k^{(1)}$ the same, and by flipping one bit in another label $k^{(2)}$. It is self-evident that $k^{(1)}$ and $k^{(2)}$ are can be obtained by one time leaving k^* as is, and one time flipping back the bit (making sure $k^{(2)}$ has bit r set, such that $k^{(i)}$ are in the correct order). Determining the amplitude factor is unproblematic as well. In

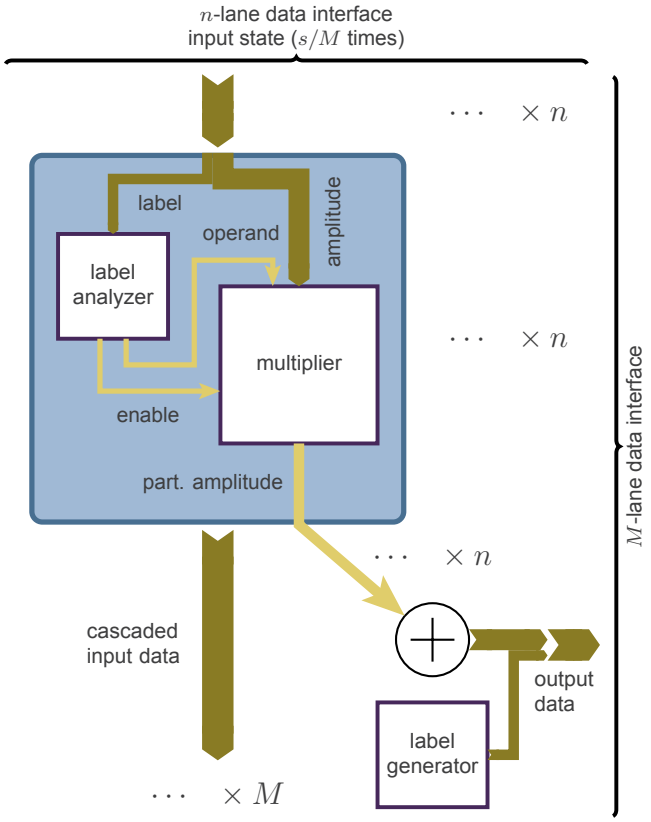


Figure 4.3: High Level scheme for embedding the *dense-gate* element update formula into a parallel, scalable computing structure. The label analyzer implements the inverse element update formula Equation (4.1), and has access to any relevant operation parameters (such as target qubit and operation matrix)

Equation (3.10), gate matrix elements m_{ij} were matched to k such that $k[r] = j$ (bit index r matches column index in m_{ij}). The desired inverse operation treats the extended operation matrix row-wise, instead of column-wise. Accordingly, the formula is adjusted such that the column row bit i matches bit index r:

$$v_i : k^*[r] \begin{cases} 0 & \rightarrow m^{*(1)}(k) = m_{00}, \quad m^{*(2)}(k) = m_{01} \\ 1 & \rightarrow m^{*(1)}(k) = m_{10}, \quad m^{*(2)}(k) = m_{11} \end{cases} \tag{4.1}$$

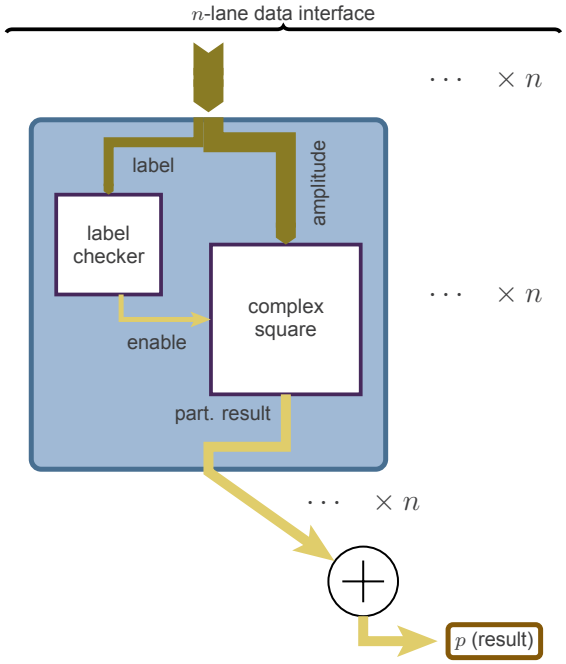
The label analyzer contains this version of the element update formula, in order to filter for required labels, and select the correct multiplicand to forward to the multiplier.

Computational Complexity

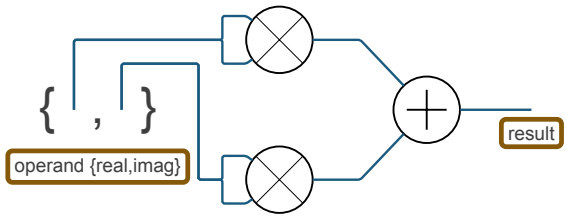
As mentioned, the method as is has quadratic computational complexity (with respect to the vector size s), instead of linear complexity (provided $s > M$). The reason is as follows (assuming s to be an integer multiple of M, for the sake of simplicity): For a vector size $s^* = u \cdot s$, the time for streaming the vector one time scales with u. For sparse gates, the discussion terminates here, hence the linear complexity. For the dense gate array, the vector additionally has to be streamed $(u \cdot s)/M$ times. Since u appears both in the time for streaming the vector one time, and in the number of streaming operations, the total computation time scales with u^2, as soon as s exceeds M. It is evident that M is restricted by the amount of available hardware, and thus by the actual hardware platform. The applicability of this array method therefore depends on the actual hardware platform, and on the state size s in the expected simulation scenarios. Next to the aforementioned alternative random writing access design, Section 7.2.5 presents a method to “artificially” increase M by means of local buffer memory.

4.3.4. Measurement Operations

As explained in Section 3.6.1, a quantum measurement operation consists of first computing an expectation value, then determining a probabilistic measurement outcome, and finally applying the outcome



(a) *measurement* probability computation (phase 1)



(b) "complex square" unit for performing Equation (4.2); notice that the input is a complex number, but the result is a real number

Figure 4.4: Embedding the measurement probability element update formula (Section 3.6.1) in parallel, scalable computing structures. The label checker implements the element update formula. The "complex square" unit computes $\alpha \cdot \alpha^\dagger$ for a complex number α.

and expectation value to the input state, in order to obtain the post-measurement state. This section highlights the hardware implementation of these steps one-by-one, with additional emphasis on the problematic inverse square-root calculation, that is necessary for the post-measurement state.

Expectation Value

Computing the measurement expectation value (or the probability for measurement outcome 1/$|0\rangle$ for qubit r) is achieved by accumulating the individual contributions of vector amplitudes that correspond to qubit r being $|0\rangle$. The contribution of a complex-valued amplitude α is $\alpha \cdot \alpha^\dagger$. Similarly to one level of the dense gate array, the label analysis and multiplication can be parallelized, with a subsequent accumulation step (Figure 4.4a). In contrast to gates, there is no output state (yet), so the CUs are not passing on any data besides the expectation value summands.

Computing $\alpha \cdot \alpha^\dagger$ can be done with only two real-number multipliers, which results in a simple hardware circuit, shown in Figure 4.4b:

$$\begin{aligned} \alpha \cdot \alpha^\dagger &= (\mathsf{real}(\alpha) + \mathsf{imag}(\alpha)) \cdot (\mathsf{real}(\alpha) - \mathsf{imag}(\alpha)) \\ &= \mathsf{real}(\alpha)^2 - \mathsf{imag}(\alpha)^2 \end{aligned} \tag{4.2}$$

Measurement Outcome

Turning the expectation value into a measurement outcome is a straightforward process: The previously computed measurement probability $p^{(1)}$ is compared to a random value a, which is sampled from a (continuous) uniform distribution in range $[0, 1]$. If $(p > a)$, the measurement result is 1 (in some implementations, it can be more accurate to use $p \geq a$ instead). Otherwise, it is 0. Considering the entropy source for the random distribution, the proposed system uses a Pseudo Random Number Generator (PRNG) based on a Linear-Feedback Shift Register (LFSR). In short, LFSRs are a standard technique in custom digital hardware, with low resource utilization, and guaranteed uniform distribution, given a long enough period. More detail is provided in Section 5.5.4.

Inverse Square-Root Calculation

In order to compute the post-measurement state, it is inevitable to compute $\frac{1}{\sqrt{p}}$ (with p being the probability for the actual measurement outcome). There is no "native" way to do this in binary hardware,

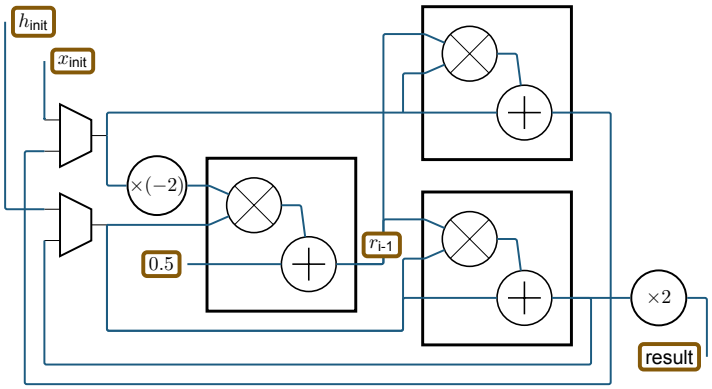


Figure 4.5: schematic for performing iterations of the Goldschmidt inverse square-root numeric algorithm in hardware (computing initial estimates x_{init} and h_{init} not depicted)

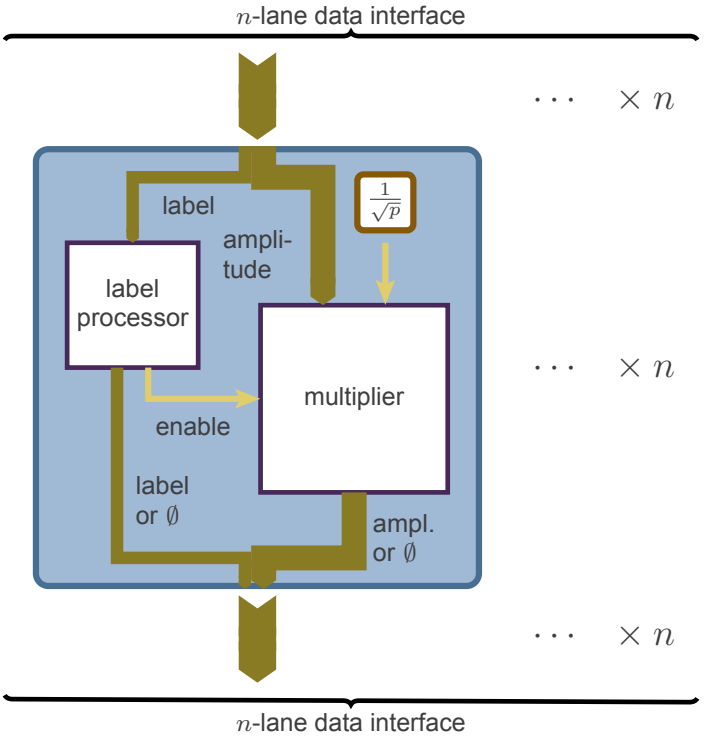


Figure 4.6: *measurement* post-measurement state (phase 2)

because it always involves a division - regardless of whether floating point or fixed-point decimal numbers are used. Floating point numbers, at least, allow “circumventing” the square-root operation by dividing the exponent by 2, but do not have an elegant solution for the division either.

Despite the lack of a hardware-efficient analytical method, there is a suitable numeric square-root algorithm, that is even applicable to both number representations: the so-called “Goldschmidt-Algorithm” [34]. It comes at the added benefit of immediately providing the “inverse square-root”, or $\frac{1}{\sqrt{p}}$, as a side-product. A block schematic, using three multiplication units for one iteration, is given in Figure 4.5. The initialization procedure to determine h_{init} and x_{init} is not discussed here, because it is implementation-dependent, and therefore belongs into the next chapter (Section 5.5.4).

Post-Measurement State

Lastly, one needs to compute the post-measurement state. The setting, depicted in Figure 4.6, is architecturally identical to the parallel array for sparse gates. The multiplier has the static second operand $\frac{1}{\sqrt{p}}$, instead of a gate matrix entry. The label processor handles the differentiation between destructive

and non-destructive measurement. When encountering labels that do not match the measurement result, it can cause the multiplier output to be 0, for non-destructive measurement, or switch off any output at all, for destructive measurement.

4.4. On-Chip Quantum State Memory

After the previous section discussed the *engine* module in Figure 4.1, this section is dedicated to the other core component in the proposed accelerator system - the *quantum state memory* module.

4.4.1. Overview

The quantum state memory unit (Figure 4.7) is designed to make maximum use of the flexibility that an FPGA offers. This shows in the fact that in the proposed system, the accelerator itself takes care of managing quantum states in the system, including memory allocation and operation. In doing so, that overhead is taken away from the host CPU, which does not have access to the accelerator's quantum states in memory at all. An additional benefit is modularity. The accelerator can implement quantum state memory in any way, up to combining different types of available memory resources, without affecting the host application, or the interface between host and accelerator. The concept is similar to how memory controllers on DRAM modules can differ in implementation, without any need for the host CPU to be aware of that. State vectors are exchanged between host and accelerator on specific request only (`set` or `readout` operation), via shared memory either on-board, or in the host's main memory. In a fitting coincidence, this design (except for the data exchange operations) resembles the interaction between a real quantum chip and a control system, because the exact states in the quantum chip as well are "blackboxes" to the control system (since qubits can only be observed via measurement).

Figure 4.7 provides the full internals of the quantum state memory module from Figure 4.1. It is divided into a *control block* and *execution block*. The control block translates the core layer interface from Table 4.1 into commands to the execution block. The components are explained bottom-up in the following sections - starting in the execution block, with the *data back end*, continuing with the *memory management* section, to finally discuss the control block's operation Finite State Machines (FSMs).

An important design aspect, that affects both control block and execution block, is that any quantum state operation uses different source and destination memory regions, and full-duplex (or dual-simplex) data buses. Prior to an operation, the result state's size is determined, and an available region in memory is allocated for the result state. After the operation, the state's pre-operation memory location is deallocated. The concept has similarities to a double buffer, but operates more dynamically, because multiple states in memory are supported (instead of for example a single image), and because input and output state can have different sizes.

4.4.2. Execution Block - Data Back End

The data back end is responsible for handling any actual data traffic in the system, supplying both the engine with state vector data, and interacting with the host system, via the front-end interface in the intermediary layer. On the memory side, the *memory interface* takes read or write requests, with allocation data like addresses and sizes, and contains the necessary bus masters to interact with the state memory. Data is multiplexed to either the engine, or the host, depending on the request. Any required bus processing or conversion for both paths is also handled by the data back end. All components are full-duplex capable - all buses are full-duplex, and there are separate, independent memory bus masters for reading and writing.

4.4.3. Execution Block - Memory Management

Memory Allocation Table

Memory management consists of the *memory allocation table*, and the *memory allocator*. Requests go exclusively to the memory allocation table, which, for every quantum state in the system, holds the required information to locate it in memory (address, size, type of memory in case of multiple memories). It is worth noting that, since this is a data structure in hardware, the table has a fixed maximum number of entries. That number can be reasonably high, depending on the type of memory that is used for the table, but the accelerator needs to be parameterized according to the target simulation scenarios at hardware compilation time.

Another consequence of the memory allocation table being an (autonomous) data structure within

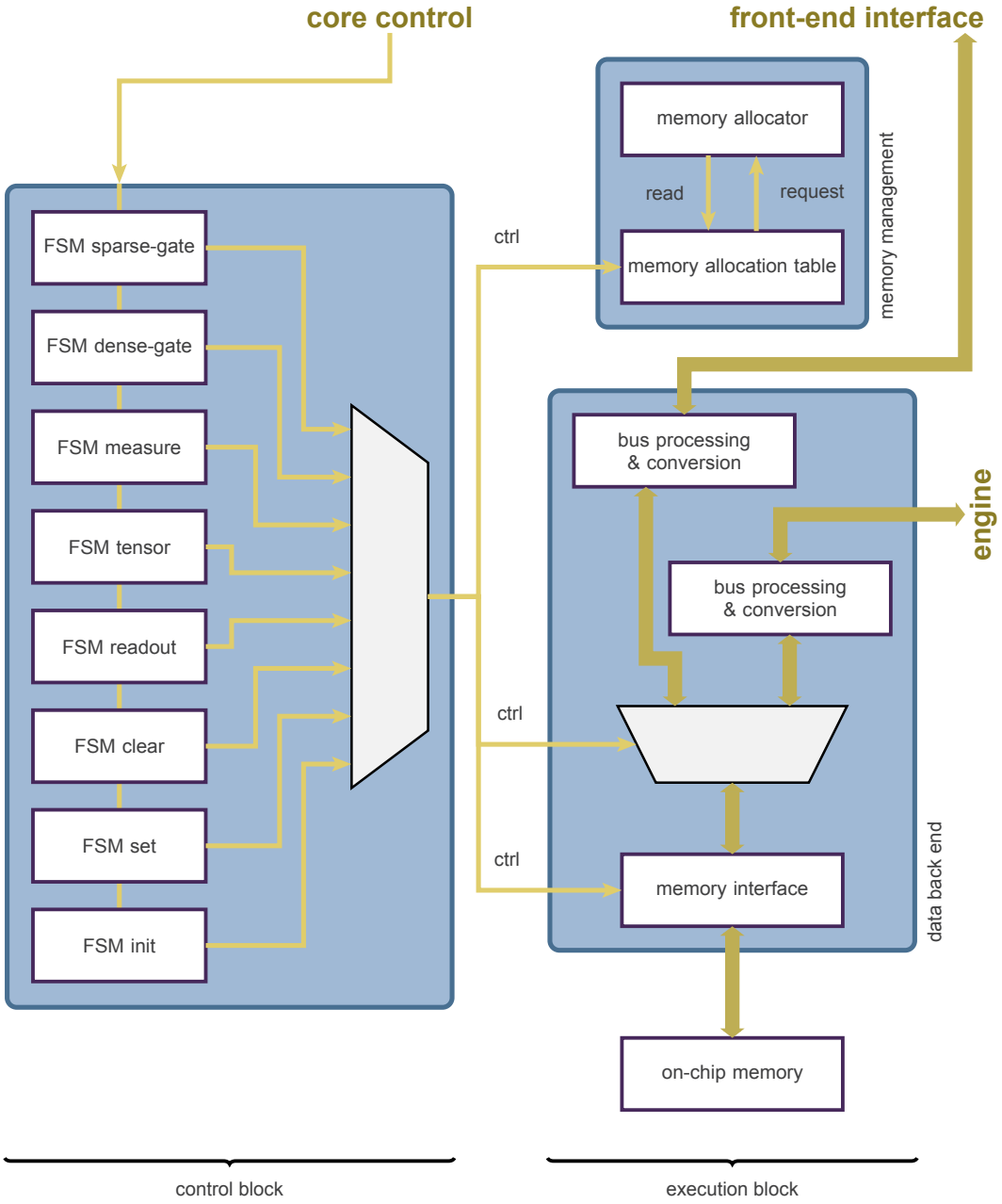


Figure 4.7: Block diagram of the *quantum state memory* module (figure 4.1). One state machine in the *control block* can be active at a time, which then takes full control of the *execution block*. The state machines have access to any relevant static operation parameters (like state index, number of qubits).

the accelerator, is that the host software has no means to directly access it. This has implications for the translation between host application qubit objects, and on-board quantum states. While a front-end application's interface could be built such that user code directly operates on quantum states, a simulator like NetSquid has an additional abstraction layer, in that user code operates on individual qubit objects (which share quantum states in the background). In this case, a qubit object first needs to be resolved to the "logical index" of its quantum state (and to the qubit's index within that state), and afterwards the "logical index" needs to be matched to the "physical index" (essentially an address) of that state in the memory allocation table. Both steps can either be performed in software, or in hardware. This thesis opts to do them in software in the prototyping phase. Therefore, software has to maintain a separate table for matching logical to physical indices, without being able to communicate with the in-hardware memory allocation table. For any operation, the software therefore needs to carefully mirror

changes to the memory allocation table, which is an easy potential error source, and creates overhead. It is recommended that a future design iteration shifts matching logical to physical state indices to the accelerator (but leaves qubit to logical state matching to software), to eliminate this error source - while still allowing the front end to either operate on quantum states, or on qubit objects.

Memory Allocator

The memory allocation table is able to manage a dynamic state space, meaning that it has the ability to remove states, and allocate new ones. The allocation process itself is handled by a separate module, the *memory allocator*, which has read access to the current allocation table. The idea here is again modularity. The memory allocator can be swapped out at any time, for either a different algorithm, or a different implementation of the same algorithm, without affecting anything else. The allocator plays an important role in the system, since it is not only needed when a new state is created, but also when an existing state requires a new allocation, or when an operation causes a state to change in size. The employed allocation algorithm depends on the available memory resources, and is therefore explained later on in Section 5.6.1.

The reason to not give the control block access to the memory allocator, and instead let the memory allocation table be the only module that controls the allocator, lies in simplifying the interface between execution block and control block. There is no case in which the allocator is needed, that is not an operation on the memory allocation table, and whenever the allocator is activated, the allocation table requires its "return object" (effectively, a memory address). It would introduce unnecessary complexity, if not a higher chance for mistakes, if the control block were to request a new allocation itself, just to then always forward it to the allocation table.

4.4.4. Execution Block APIs

The interface between control block and execution block consists of a set of pre-defined commands, that address either the memory allocation table, or the memory interface (including the data bus multiplexer). Since memory read and write are not always synchronous, the memory interface's read and write channels are controlled separately. In software terms, the respective read and write commands could therefore be called non-blocking, indicating that a control FSM can initiate a read or write, and then move on. As a convention however, in order to reduce the amount of `wait_*` statements in this and the following paragraphs, activating a subsequent read or write operation (respectively) still waits for the previous one to be finished.

- **memory allocation table**
  - `mem_alloc_table_get(state index)`
    Retrieve the allocation data (address, size, physical memory) for a quantum state.
  - `mem_alloc_table_update(state index[, size])`
    Allocate a new memory location for a quantum state (used when performing a quantum operation, to determine a suitable output state memory location). Optionally, change the state's size for the new allocation (for tensor operations and destructive measurement).
  - `mem_alloc_table_clear()`
    Clear the memory allocation table ("resetting" the system).
  - `mem_alloc_table_req_new(size)`
    Find an allocation, and create a new table entry for a state of `size` qubits.
  - `mem_alloc_table_remove(index)`
    Remove a state from the allocation table.
- **memory interface**
  - `mem_intf_init(allocation)`
    Initialize a state $|0...\rangle$ at `allocation` (which contains both memory location, and size)
  - `mem_intf_read_nblk(allocation, client)`
    Read state from `allocation`, to `client` (engine or front-end interface/host). Non-blocking (but waits for an ongoing `read` to finish).

- `mem_intf_write_nblk(allocation)`
  Write to `allocation`, from `client` (engine or front-end interface/host). Non-blocking (but waits for an ongoing `write` to finish).
- `mem_intf_wait_read()`
  wait for an ongoing `read` to finish
- `mem_intf_wait_write()`
  wait for an ongoing `write` to finish
- `mem_intf_wait_idle()`
  wait for ongoing `read` and `write` to finish

4.4.5. Control Block

The control block's task is to implement the core layer interface from Table 4.1, using the execution block APIs introduced in the previous section. In a software design, these would be sub-routines. In a hardware design, they translate into individual, procedural FSMs. This section describes the operation of the state machines in an intuitive, software-like manner, as algorithms that consist of elementary control structures, and the execution block instruction set.

State Space Interaction

Algorithm 1 clear system (remove all states)

```
1: procedure Clear
2:     mem_alloc_table_clear()
3: end procedure
```

Algorithm 2 create new state with n qubits

```
1: procedure Create(n)
2:     entry = mem_alloc_table_req_new(n)
3:     mem_intf_init(entry)
4: end procedure
```

Algorithm 3 set state s from host application

```
1: procedure Set(s)
2:     entry = mem_alloc_table_get(s)
3:     mem_intf_write(entry, client=host)
4: end procedure
```

Algorithm 4 read out state s to the host application

```
1: procedure Readout(s)
2:     entry = mem_alloc_table_get(s)
3:     mem_intf_read(entry, client=host)
4: end procedure
```

Gate Operations

Algorithm 5 perform a sparse-gate operation on state s

```
1: procedure Sparsegate(s)
2:     entry_read = mem_alloc_table_get(s)
3:     entry_write = mem_alloc_table_update(s)
4:     mem_intf_write_nblk(entry_write)
5:     mem_intf_read_nblk(entry_read)
6:     mem_intf_wait_idle()
7: end procedure
```

Algorithm 6 perform a sparse-gate operation on state s

```
1:  procedure Densegate(s)
2:      entry_read = mem_alloc_table_get(s)
3:      entry_write = mem_alloc_table_update(s)
4:      num_subtransmissions_read =
5:          ceil(entry_read.size / engine_densegate_array_depth)
6:      mem_intf_write_nblk(entry_write)
7:      for n in range(num_subtransmissions_read) do
8:          mem_intf_read_nblk(entry_read)
9:      end for
10:     mem_intf_wait_idle()
11: end procedure
```

Non-Gate Operations

Algorithm 7 perform a measurement (destructive or in-place) on state s

```
1:  procedure Measure(s,destructive)
2:      entry_read = mem_alloc_table_get(s)
3:      mem_intf_read_nblk(entry_read)
4:      if destructive then
5:          num_qubits_post = entry_read.num_qubits - 1
6:      else
7:          num_qubits_post = entry_read.num_qubits
8:      end if
9:      if num_qubits_post == 0 then
10:         mem_alloc_table_remove(s)
11:     else
12:         entry_write = mem_alloc_table_update(s, num_qubits_post)
13:         mem_intf_write_nblk(entry_write)
14:     end if
15:     mem_intf_read_nblk(entry_read)
16:     mem_intf_wait_idle()
17: end procedure
```

Algorithm 8 merge state s_2 into s_1 ($s_1 \otimes s_2$)

```
1:  procedure Tensor(s1,s2)
2:      entry_read_reg = mem_alloc_table_get(s1)
3:      mem_intf_read_nblk(entry_read_reg)
4:      entry_read_stream = mem_alloc_table_get(s2)
5:      num_qubits_post =
6:          entry_read_reg.num_qubits + entry_read_stream.num_qubits
7:      entry_write = mem_alloc_table_update(s1, num_qubits_post)
8:      num_batches = ceil(entry_read_reg.size / engine_tensor_num_regs)
9:      mem_intf_wait_read()
10:     mem_intf_write_nblk(entry_write)
11:     mem_intf_read_nblk(entry_read_stream)
12:     for i in range(num_batches-1) do
13:         mem_intf_read_nblk(entry_read_reg)
14:         mem_intf_read_nblk(entry_read_stream)
15:     end for
16:     mem_intf_wait_idle()
17:     mem_alloc_table_remove(s2)
18: end procedure
```

4.5. GPU Applicability

As indicated earlier in this chapter in Section 4.2, a considerable portion of the presented computation architecture is not only suitable for an FPGA, but also for a GPU. Additionally, it was pointed out that in certain scenarios, a GPU might even offer performance advantages over an FPGA. Since GPUs are arguably the most widespread type of accelerator platform, this section provides a brief study on porting this chapter's design to a GPU platform.

4.5.1. Analysis

The following paragraphs elaborate on feasibility and potential problems in a top-down wap, layer-by-layer, referring to the design overview in Figure 4.1. For the core layer, the engine and the quantum state memory module are discussed separately.

GPU Applicability - Front End Layer

From an architectural point of view, the front end layer remains unchanged. The low-level software modules interacts with the GPU via the respective framework (for instance, CUDA or OpenGL), and provides the abstract core layer API (Table 4.1) to a user application of choice.

GPU Applicability - Intermediary Layer

The intermediary layer is practically eliminated, since its task is bridging between low-level software and accelerator kernel functionality. A GPU, driven by a suitable software framework, provides that functionality natively.

GPU Applicability - Core Layer - Engine

Most of the engine's architecture translates well to a GPU. The engine's individual modules are composed of parallel PEs, with parallel input and output data streams. These PEs perform relatively simple, dense computations. Both the dataflow pattern and the computational requirements match the general architecture of a GPU. The main difference between an implementation in custom digital logic, like FPGAs, and a GPU is that in the former case, the individual engine modules are most likely realized as separate hardware modules, meaning that all of them are physically present in the accelerator. In a GPU however, the individual operation modules (sparse gate, measure, tensor, . . .) would rather be prepared as GPU kernels, and loaded to the device dynamically by the host CPU.

Several kernels, or engine modules, require a more thorough elaboration, because they deviate from the simple model of fully independent PEs, with parallel input and output data streams:

- Dense Gate Module
  - The data stream is not fully parallel, since all PEs require the full input vector (instead of being assigned one "portion" of the vector to handle). Although it remains to be studied how efficiently a GPU allows forming CU arrays with propagating input data, compared to an FPGA, this dataflow pattern does not pose a general issue to the GPU
  - The dense gate module additionally requires accumulation across sets of CUs (across "array levels", in Section 4.3.3). This can be achieved on GPUs via either manual or native accumulation.
- Tensor Module
  - The tensor module requires local buffers for the registered state ("state 2" in Section 4.3.2). Since PEs in a GPU feature (shared) local SRAM buffers, they fulfill this requirement. Shared buffers in this scenario are in fact an advantage, rather than a hindrance, because all PEs require the same buffered data, in the same order.
- Measurement Module
  - The measurement module requires an RNG in order to determine the measurement outcome from a probability, or expectation value. Although not necessarily an efficient operation on a GPU, viable on-device approaches exist, which stem from seeding Monte Carlo simulation in applications like financial projections (see for instance [35], ch. 37, for a discussion around Nvidia's CUDA).

  - Additionally, the measurement module requires an inverse square-root computation, for which the Goldschmidt-Algorithm has been proposed as an in-hardware solution for custom digital logic. In theory, the same algorithm is also applicable to a GPU. However, Nvidia CUDA for instance provides a native API for reciprocal square-root computation. Further study is required to determine the most practical method, while usually the native API is the preferred solution.

GPU Applicability - Core Layer - Quantum State Memory

The majority of the quantum state memory module is either handled by internal GPU hard-IP. The remainder is shifted into the host CPU, as the following analysis shows (referring to Figure 4.7):

- The tasks of the *memory management* module (determining and handling accelerator main memory allocations) most likely needs to be handled by the host system, in the low-level software layer. All required addresses are usually determined before the host CPU invokes a GPU kernel. In terms of feasibility, this does not pose an issue. In terms of performance, while there might be an impact, it is not considered to be problematic, because the data structures for memory management are relatively small, and because no communication is required between host and accelerator.
- The *data back end* module is likely to be removed, because memory interfacing and data distribution is done natively by the GPU.
- The *control block* becomes obsolete, because its purpose lies in orchestrating the operation of the memory management module, the data back end, and partially the engine. As explained, the memory management functionality would not reside in the accelerator, and the explicit data back end be removed. The engine is fully controlled by the compute kernels, issued by the host system. Therefore, all responsibilities of the control block FSMs are taken over by the native GPU system, or the host CPU.

4.5.2. Summary

In summary, this section demonstrated that the proposed design is feasible to be implemented on a GPU as well, instead of an FPGA, with very few changes to the functional *core* layer, and the same user application API. The most relevant adaptations concern removing design units that are replaced by native GPU functionality, implementing accelerator memory management in low-level software, and replacing non-parallel computation steps like random number generation and inverse square-root with GPU-optimized alternatives.

4.6. Conclusion

This chapter proposed a parallelizable, high-level design for a hardware accelerator, built around labeled-random state vector representation and element update formulas.

In Section 4.1, a comparison was performed between the advantages and disadvantages of implementing labeled-random state vector computation on a multi-core CPU, a GPU, or an FPGA. A multi-core CPU system turned out to be an inefficient choice, that would lack scalability in both computation parallelization and memory bandwidth. GPUs and HPC FPGAs in contrast both appeared to be suitable platforms, with tradeoffs between the two in computation power and extensibility, with respect to different simulation scenarios. An FPGA was chosen for this work, mainly due to the greater flexibility, which allows the prototype to serve as a starting point for design space exploration into various directions.

Sections 4.2 to 4.4 presented a modular accelerator system, that fully spans from quantum states in on-board memory, to a swappable quantum simulator application front end on a host CPU. Any implementation-specific aspects were left to the chapter hereafter, which allows for the discussion in this chapter to apply to a majority of custom digital hardware platforms. The focus laid on the “functional” computation components. A computing engine, consisting of parallelizable multiplier-based CU arrays, was paired with a data back end, which exploits the flexibility of an FPGA to provide full on-chip quantum state memory management. The design relies on clear operation interfaces in between modules and layers, which enables seamlessly swapping in and out components. This greatly simplifies future design iterations, as well as matching the accelerator back end to a front-end simulator’s needs.

Finally, Section 4.5 studied the applicability of the accelerator system to GPUs, instead of custom digital hardware, for a potential future accelerator porting. It was found that the computational hardware

kernels translate well to GPU kernels, with several adaptations to non-parallel computation steps. The remainder of the system was found to be considerably simplified in a GPU system, because many control and data handling functionalities would be taken over by native GPU hard-IP, with only small additional overhead for the host CPU.

5

FPGA Implementation

The previous chapter introduced the proposed accelerator system design at system level, in an implementation-agnostic way. Consequently, this chapter is dedicated to implementation level. The starting point for the hardware implementation lies in defining a target system. Section 5.1 discusses all related aspects, from FPGA platform choice, through implementation guidelines, to front-end simulator. Afterwards, Section 5.2 outlines the implementation scope of the prototype that is delivered with this thesis, to give a clear distinction between proposed design, and hardware-validated prototype.

Conceptually, analogous to the system-level discussion in Chapter 4, the remainder of the chapter highlights the various hardware and software system components (Figure 4.1) in a bottom-up fashion, with a focus on implementation-specifc aspects. With the choice for an implementation platform however, and the identification of performance targets, come several “universal” aspects of an implementation. Examples that are important to this particular application are number representation and arithmetic circuits - because the system needs to handle large amounts of complex and real number computation - and data buses - because data movement between memory and CUs is a crucial part of the system. Since these universal design parts are not tied to one specific component, they are established first (Sections 5.3 to 5.4), before Sections 5.5 to 5.8 tackle individual modules.

The chapter closes with an overview of the parameterization for the implemented and validated prototype, at the time of delivering this thesis (Section 5.9).

5.1. Target System

This section establishes the framework for implementing the presented accelerator design from Sections 4.2 to 4.4 in a “demonstration system” or prototype, using a specific FPGA accelerator system. Firstly, Section 5.1.1 reasons about the chosen target FPGA platform. Afterwards, Section 5.1.2 discusses important implementation metrics and target figures - with respect to the accelerator design and FPGA platform - to guide FPGA-specific design decisions. Finally, Section 5.1.3 introduces the front-end *user application* (see Figure 4.1) that is targeted for the demonstration system.

5.1.1. FPGA Platform

This work uses the AMD/Xilinx Alveo U55C Data Center accelerator card [36]. Different from the process of choosing an accelerator platform type in Section 4.1, which was based on a comparison between various options, the AMD/Xilinx Alveo product range was selected as the starting point, before formulating criteria. The main reasons lay in the fact that these devices form the most widely distributed explicit FPGA accelerator platform, which guarantees availability, scalability, and a high degree of toolchain familiarity. Nevertheless, this initial choice was still evaluated against a series of criteria, with respect to labeled-random state vector computation.

Criteria

Based on the accelerator high-level design from Chapter 4, the following criteria for a suitable target FPGA are derived:

- Availability

  Given the generic simulator back end, easy deployment with various front ends is instrumental, which requires hardware availability. The same holds for expanding an existing installation, either via multiple separate back end instances, or via clustering.
- Extensive on-chip computation capabilities
- High-bandwidth on-chip or closely coupled memory
- Good front-end software integration
- Accessibility

  The design should facilitate future extensions, additions, and maintenance, by various contributors, which is closely coupled to general familiarity with the programming framework and toolchain.
- Extensive local on-chip memory resources
- Peer-to-peer networking

Evaluation

It was found that the U55C fulfills these criteria:

- Availability: The U55C is commonly seen in data centers, as the effective successor of the U280, and the most common HBM-equipped AMD/Xilinx Alveo accelerator card.
- Extensive computation capabilities: The chip offers 9024 DSP slices, $1{,}304{,}000$ LUTs, and $2{,}507{,}000$ registers.
- High-bandwidth on-chip or closely coupled memory: $16\,\mathrm{GB}$ HBM are available.
- Good front-end software integration: The Vitis compiler toolchain provides a native workflow, designed for software integration via the XRT driver library.
- Accessibility: The ecosystem is generally well-known because AMD/Xilinx is the world-leading FPGA vendor, although the less established Vitis compilation flow bears some potential hindrances.
- Extensive local on-chip memory resources: The chip is equipped with roughly $42\,\mathrm{MB}$ on-chip (of which $33.75\,\mathrm{MB}$ UltraRAM, rest BRAM).
- Peer-to-peer networking: 2 QSFP28 ports with up to $100\,\mathrm{Gbit/s}$ each are available.

In addition, the U55C was compared to the other HBM-equipped accelerator cards available from AMD/Xilinx (U50 and V80). The U55C has better availability, and is the best fit in terms of sizing and architecture. The U50 is significantly smaller, while the projection is that the design can take advantage of the U55C's additional resources, and potentially of the native ring-topology capability that comes with the second serial $100\,\mathrm{Gbit/s}$. The V80's architecture was considered unfavorable, because the Versal System on Chip (SoC) introduces unnecessary complexity, there is no immediate use for the additional DDR4 memory, and the U55C's higher DSP-to-LUT ratio lends itself better towards a multiplication-heavy application.

5.1.2. FPGA Implementation Targets

For implementing the accelerator design on the U55C FPGA (or an FPGA in general), it is important to define target figures, with respect to typical metrics such as resource consumption, power consumption, and computational throughput. These guide central design decisions, among which data bus implementation, and pipelining support. As an example, an ultra low-power application can sacrifice clock frequency, and thus computational throughput, for lower power consumption, and might opt for entirely different bus protocols than an HPC accelerator. This section examines the application and platform at hand, with respect to the above implementation metrics. First, an application-driven prioritization is formulated, followed by a quantification, based on feasibility considerations in the U55C hardware.

Prioritization

A quantum simulator back end's leading purpose is to reduce the computation time for quantum state operations. In terms of implementation metrics, the first implication thereof is that power or resource consumption are of secondary concern - as far as physical card limits allow - and should not compromise computational throughput. Since the U55C is a dedicated accelerator card, it is designed for physical

card limits that allow high power consumption, such that power was not considered in any design decision.

"Reducing computation time" can, in general, either mean reducing latency, or increasing throughput. In the given application, it refers to throughput, because quantum simulation processes data structures that grow exponentially in size, which usually causes computation throughput to dominate computation latency. Since FPGAs can not reach the clock frequencies that ASICs are capable of (in the GHz range), they require parallelization and extensive pipelining for high throughput - combined with a reasonably high clock frequency, within the device limits. One way in which FPGAs accommodate this approach is the practically unlimited availability of pipelining registers. Since there are roughly two registers per LUT, almost any design runs out of LUTs or becomes unroutable, before it runs out of registers. Additionally, hard-IP like the DSPs are built with multiple internal pipeline stages. Summarizing, the accelerator system needs to be designed to support a high throughput via large-scale pipelining, at reasonably high clock frequencies, with resource consumption playing a secondary role, and minimal attention to power consumption.

Quantification and Guidelines

In order to quantify "reasonably high clock frequencies", it helps to look into the potentially involved hard-IP components. Most importantly, these are the on-chip DSPs and HBM. While the respective data sheets give theoretical absolute maximum operating frequencies, a potentially "more realistic" alternative is looking into a highly optimized design. Xilinx application note 1332 [7] is a demonstration project published by Xilinx for DSP-based matrix-vector multiplication, which is similar to this accelerator in terms of parallel and potentially cascaded multiplication processing. That project achieves a clock frequency of $670\,\mathrm{MHz}$ at a hardware utilization of roughly $70\,\%$, across 3 SLRs, on Xilinx UltraScale+ hardware (same technology and primitives as the U55C). $670\,\mathrm{MHz}$ therefore is a good figure for a realistic DSP operation frequency ceiling. The HBM itself is not routing-dependent, such that the exact maximum figures from the hardware documentation apply [37]. The memory's fabric logic interface can not operate any faster than $450\,\mathrm{MHz}$ (which also maximizes the memory throughput), because the internal DDR memory controllers operate at a maximum of $900\,\mathrm{MHz}$. Therefore, $450\,\mathrm{MHz}$ is the desired operating frequency for all design modules that are involved with computation, to match the maximum memory bandwidth. It is projected that even more complex modules, whether or not involved in the computation, eventually can be built to reach this frequency, since $450\,\mathrm{MHz}$ evidently leaves some margin to realistic operating limits. Therefore, it is estimated that there is no need for considering a multi-clock domain system.

Next to pipelining, several other design techniques help with achieving high clock frequencies, which are employed in the design wherever applicable. Among these are avoiding high-fanout paths, reducing logic depth where a high fanout is inevitable, and keeping logic in datapaths instead of control paths (see [38], chapter 3, section "RTL Coding Guidelines").

5.1.3. Front-End Layer

Next to a specific target FPGA, the proposed prototype system includes a target user application, thus a target front-end simulator. The idea here is not to shape the accelerator towards a specific simulator, but rather capitalize on using an already existing front end - which saves implementation work, and provides a proven (CPU-based) simulator to serve as a validation system and performance baseline.

The NetSquid simulator (introduced in Section 2.4.3) is a suitable target simulator for various reasons:

- It is proven across various experiments and publications, and can therefore be considered reliable.
- In terms of software architecture, it lends itself well towards integrating an accelerator system:
  - It has a modular quantum state representation back end, which provides a convenient integration point for the accelerator back end (although potentially not ideal from a performance standpoint, see Section 6.3.6)
  - The lower layers are written in cython, which allows for importing C/C++ libraries, while XRT software is written in C++, with native support for compiling into a Dynamic Link Library (DLL).
- It is designed as a Quantum Network simulator, such that it can fully use the accelerator's functionality set (in contrast to a pure Quantum Computing simulator). It is still possible for NetSquid to act as a pure Quantum Computing simulator, by ignoring its intended protocol-level

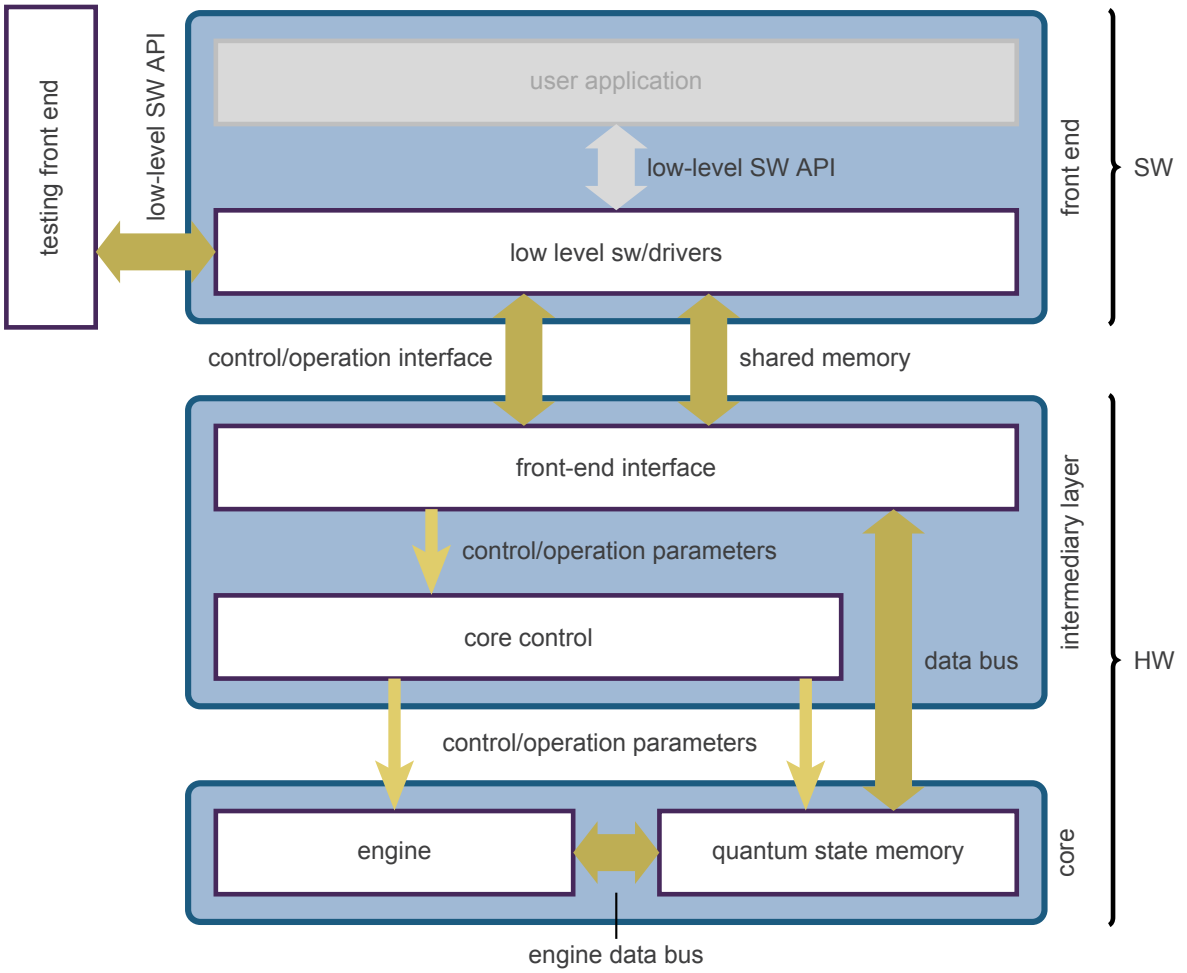


Figure 5.1: implementation scope (in relation to full system design, figure 4.1). instead of a *user application*, a *testing front end* is implemented, using the same *low-level SW API*, that supports both validation and benchmarking/profiling.

simulation API, and directly using its qubit API instead. That allows it to also showcase the accelerator in more "homogeneous" simulation scenarios.

- The proximity to the NetSquid user base at QuTech allowed to use the same HPC node for performance evaluations as their publications do, which guaranteed meaningful baseline data.

5.2. Implementation Scope

The previous section provided a thorough discussion of the eventual FPGA implementation target - a fully functional prototype, optimized for computational throughput, on an AMD/Xilinx Alveo U55C platform, integrated into NetSquid as the front-end user application. Within the scope of this thesis however, the final step of connecting low level software and user application (Figure 5.1) needed to be left out. The following paragraph gives a functional overview of the delivered project described in this thesis, with respect to Figure 4.1.

Up to the low level software layer, a fully implemented and functional system delivered with this thesis. Components like the quantum state memory module are kept relatively straightforward, compared to the possibilities of the U55C FPGA, in favor of presenting a working prototype. A detailed picture of this current demonstration system status is given at the end of this chapter, in Section 5.9. Considering the user application, it is made sure that the implemented low level software layer is compatible with the NetSquid simulator. For this thesis, a *testing front end* has been implemented and integrated instead. The testing front end contains functionality for operation validation, and for performance evaluation. It connects to the same low level software API that a future NetSquid-specific "software bridge" connects to.

5.3. Number Representation

After the previous sections established the framework for the implementation, and provided an outlook on thesis scope, this section initiates the discussion of the actual FPGA implementation. This first portion focuses on on-chip number representation of quantum state vectors. The reason for separating that topic from the subsequent sections is that it is a system-wide decision, with effects on functional units across the entire FPGA design. For this reason, the format of labeled-random quantum state vectors in the proposed system is defined in this section, and henceforth treated as immutable. In order to derive a suitable number representation, the characteristics of quantum state vector amplitudes need to be understood, the required mathematical operations considered, and the FPGA's computation resources analyzed.

Quantum state vector amplitudes are complex numbers, with continuous absolute values (or radius, in polar coordinates) in the range $[0, 1]$. The element update formulas require the ability to perform a large number of both multiplications and additions on these amplitudes. The (theoretical) degrees of freedom are whether to represent complex numbers in polar or in cartesian coordinates, whether to use floating-point or fixed-point real numbers, and the bitwidths of the representation's components (integer and fractional part in fixed-point, mantissa and exponent in floating-point).

Complex Numbers

The answer to the complex number representation is dictated by the required operations: Although multiplying complex numbers in polar coordinates takes less individual multiplications than in cartesian coordinates, they have the disadvantage that addition essentially requires conversion to cartesian coordinates. Since both operations are needed, the accelerator uses cartesian coordinates exclusively.

Real Numbers

The question of real number representation type also has a straightforward answer. The FPGA has hard-IP DSPs, which are built for two's complement integer multiplication and addition. In terms of these operations, fixed-point real numbers handle no different from integer numbers. The only deviation lies in the bit interpretation, or in simple language, "where one sets the decimal point". Floating-point numbers in contrast cost considerably more hardware in FPGAs, and can additionally lead to slower hardware designs because of the extensive fabric logic circuitry. Therefore, the accelerator internally uses fixed-point two's complement real numbers.

In the given application, fixed-point numbers have the additional benefit of offering a high "information efficiency per bit", when deciding about the number of integer and decimal bits. This refers to two aspects. First, amplitudes are mathematically guaranteed to not exceed an absolute value of 1. Therefore, only 1 integer bit (next to the obligatory sign bit) is sufficient. The remaining bits can be spent on decimal precision. The most logical decision about the total bitwidth is to follow the chip's DSP resources. The Alveo U55C has an FPGA fabricated in Xilinx's UltraScale+ technology, with the so-called DSP48E2 DSP primitives (see Section 2.1.3). These support a multiplication operand bitwidth of up to $27\,\mathrm{bit}$, which is hence the real number bitwidth in the accelerator.

The second argument on "information efficiency per bit" lies in the fixed intervals between fixed-point numbers, in contrast to the dynamic range of floating-point numbers. Within the same fixed-point representation, the difference between two adjacent numbers is always the same. For floating-point numbers, that distance on average increases exponentially, the larger the number gets in absolute value. As a result, floating-point amplitudes close to 0 have significantly higher precision than amplitudes close to $1/-1$. Both ends of the scale however are equally important, especially in noisy quantum simulations, which is what labeled-random state representation is designed for. This does not necessarily point towards a problem in floating-point systems, partially since there often is sufficient bitwidth for the effect to be neglectable, at such small number ranges. The crucial argument here is that a comparably small bitwidth (comparing $27\,\mathrm{bit}$ to the commonly used IEEE $64\,\mathrm{bit}$ floating-point format), in a fixed-point implementation, can still yield good accuracy, due to "efficient" use of every bit, in any situation.

Labeled State Vector Elements

In labeled-random state representation, every amplitude is accompanied by a label, indicating its index in the state vector. The label is evidently an integer number. An FPGA implementation provides the freedom to fine-tune integer number bitwidth to the desired number range. How large a state vector element label can become, depends on how large the state can become. The maximum supported state

size on the accelerator is determined by the maximum state size that fits into the quantum state memory. This prototype uses $8\,\mathrm{GB}$ of on-chip memory for quantum states, with single elements aligned to $128\,\mathrm{bit}$ boundaries for efficient accessing and address calculation. This allows for one state of up to 29 qubits:

$$\begin{aligned}\text{vector element: } & 128\,\mathrm{bit} = 2^7\,\mathrm{bit}\\ \text{memory: } & 8\,\mathrm{GB} = 2^3 \cdot 2^{30} \cdot 2^3\,\mathrm{bit} = 2^{36}\,\mathrm{bit}\\ \text{state size: } & \log_2(\frac{\text{memory}}{\text{vector element}}) = \log_2(\frac{2^{36}}{2^7}) = 29\end{aligned}$$

For this reason, this prototype uses $29\,\mathrm{bit}$ amplitude labels.

Concluding, one state vector element therefore consists of one $29\,\mathrm{bit}$ unsigned integer label, and one complex amplitude in cartesian coordinates. The complex amplitude in turn is composed of $27\,\mathrm{bit}$ two's complement fixed-point real and imaginary components, with each 1 sign bit, 1 integer bit, and 25 decimal bits.

5.4. Backbone Implementation Components

After the previous section started the technical discussion on the FPGA implementation with the most "global" aspect - state vector number representation - this section continues the approach of going from "global to individual units". In the proposed design, several components in the *core* layer (Figure 5.1) appear in multiple individual units, which hierarchically puts them in between of "individual units" and "global" (since they widely apply to the core layer, but not to the intermediary layer). This section discusses such in-between components, before delving into individual units.

Namely, the two design portions to be introduced in this section are a generic complex number multiplication unit, and a low-level on-chip data bus. Complex number multiplication, with slightly different dataflow, is a recurring task across the computation engine. Section 5.4.1 explains the "blueprint" of all complex number multiplication units in the accelerator, such that the individual units of the engine can either directly use that circuit, or build upon it. A low-level data bus is required to connect engine and quantum state memory module. Section 5.4.2 elaborates on the reasons for employing a custom data bus, and presents its functionality.

5.4.1. Complex Number Processing with DSPs

As the high-level circuits in Section 4.3 showed, a key component across the engine is a multiplier that can handle complex numbers. The internal number representation was designed such that the engine can fully utilize the available DSP slices in the FPGA, in order to form these multipliers. The circuitry for the implementation's *standard complex number multiplication unit*, or *complex DSP multiplier*, is given in Figure 5.2, and will be explained in the following. It is worth noting that some engine modules might not employ this exact circuit, but any complex number computation in the system uses at least a derivative of this design.

Multiplying two complex numbers takes up 4 DSPs, which mimic the 4 multiplication terms of a complex number in cartesian coordinates. Figure 5.2 shows one half of that. Since one multiplier input supports only $18\,\mathrm{bit}$ operands, one number is always shortened to 18 bits, by truncating the 9 least significant bits (thus reducing the number precision). It is decided per operation which number is "less harmful" to be truncated. The complex multiplier leverages that both real and imaginary part of the result are obtained from a 2-operand addition, by coupling the results of 2 adjacent DSPs via the native output cascade routing. The "lower" unit in the figure compensates for the cascading latency cycle (the P register) by disabling one A/B input pipeline stage. For clock frequency reasons, it was considered advantageous to at least keep 1 input register (otherwise, the multiplier would become part of the fabric logic timing path to the DSP). Therefore, the "upper" DSP has to enable the full 2-cycle input pipeline. The post-multiplier registers (M) and the output registers (P) are all enabled, again for timing reasons (it was not determined whether or not that is required to reach the intended $450\,\mathrm{MHz}$).

5.4.2. Engine Data Bus

The proposed system uses a custom, lightweight data bus, for the connection between engine modules and quantum state memory. The reason is twofold: The bus is first designed to keep the control overhead in the engine modules at a minimum, because the overhead could scale with the engine itself,

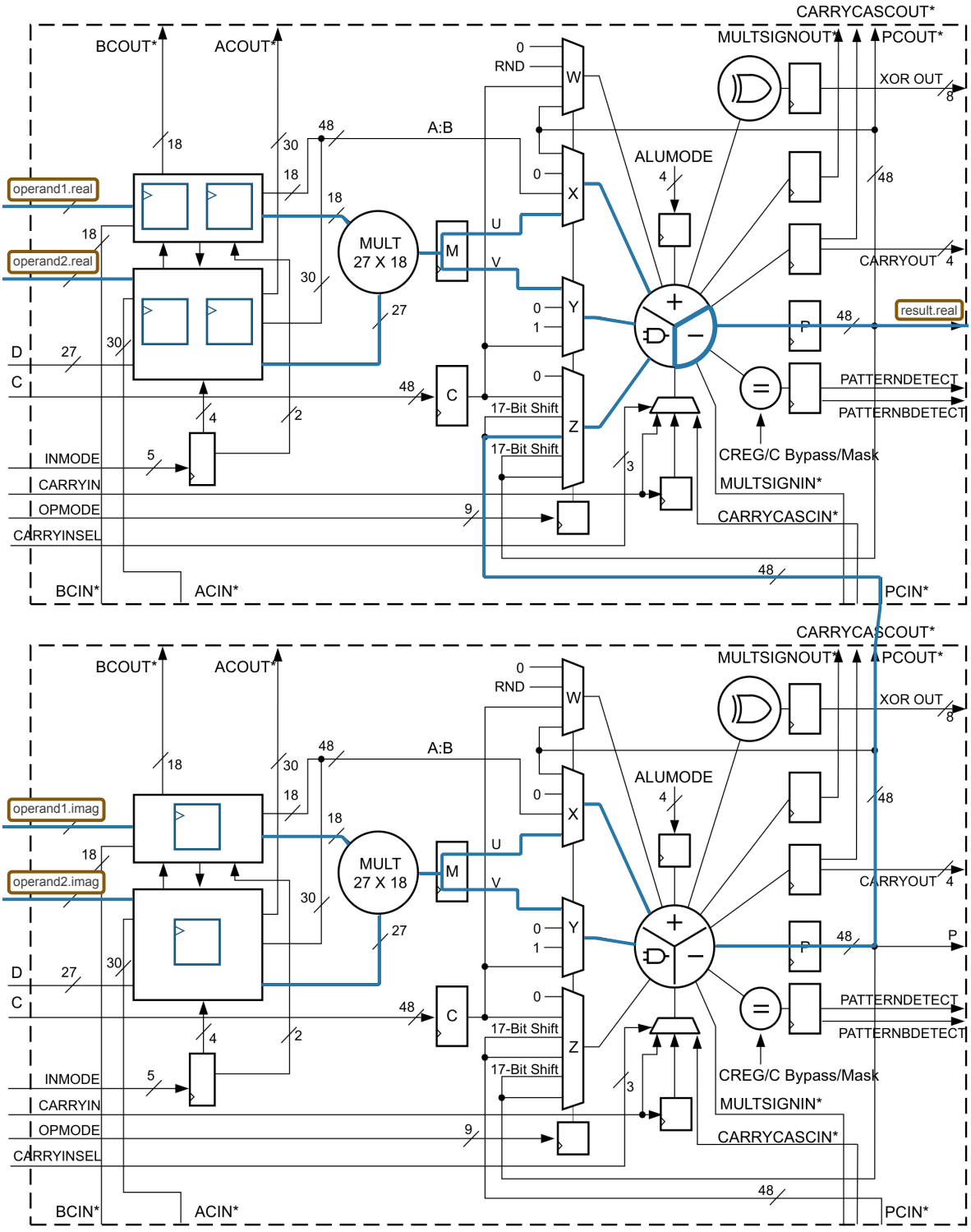


Figure 5.2: "Real" half of the DSP *complex DSP multiplier* (based on Xilinx DSP48E2 schematic [5]), computing the real portion of the complex result. Upper ALU operation is $(X + Y - Z)$ (simplified) to compensate for $j * j = -1$. "Imaginary" half is built equally, with upper ALU configured to $(X + Y + Z)$.

which follows memory interface scaling. The second reason is that endpoints for data movement are expected to be relatively scattered across the chip, because the multiplexed engine modules spatially stretch across the entire chip. A complex databus could therefore cause spiking hardware utilization, and hinder routability. The following paragraphs first introduce the bus protocol (henceforth called

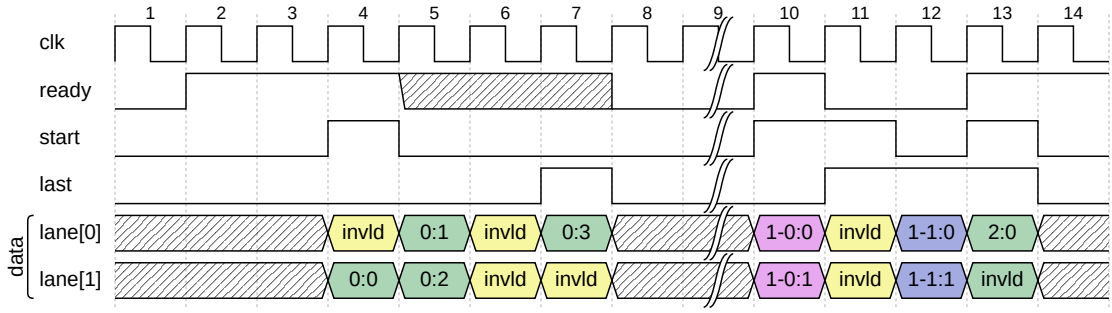


Figure 5.3: transmissions without (cycle 4-7, cycle 13) and with (cycle 10-12) subtransmissions on the *engine data bus*; data item indexing: `<transmission>[-<subtransmission>]:<item>`

engine data bus), and then elaborate on the aforementioned design reasons. AXI Stream is used as a reference, since it would have been an intuitive choice for point-to-point parallel data connections.

Engine Data Bus - Protocol

The *engine data bus* is a point-to-point, master-slave, parallel data bus, just like AXI Stream. It is described here as a unidirectional bus, with master transmitting data to slave, to be consistent with the implementation code. However, two simplex connections are used between engine and quantum state memory (input and output), such that they form a bidirectional, dual-simplex bus, with independent read and write channels. Conceptually, the bus differs from AXI Stream in that it does not have "per data-element" handshaking. Instead of individual data items, the engine data bus only operates on "transmission" granularity (one could also say "packets"). In terms of signals, the bus has a variable number of data lanes (which each transmit one vector element per cycle), and a control set consisting of the signals `ready`, `start` and `last`. The data lanes have a means of "invalidating" a vector element field, like AXI's `strb` field - either via a per-lane `valid` signal, or (as preliminarily done in this implementation) by a reserved element label.

A transmission on the engine data bus is depicted in Figure 5.3, with the respective explanation in this and the following paragraph. A slave sets the `ready` signal to indicate that it is ready to receive a transmission. From that point on, the slave is not allowed to deassert the `ready` signal, until the start of a transmission (the signal is meaningless during a transmission). The master starts the transmission by setting the `start` signal, and ends it by setting the `last` signal. The duration of the transmission, and when or how much actual "valid" data is transferred, is entirely up to the master (mostly assuming that both sides know enough about the transmission, that it is not necessary to exchange any meta information via the bus).

An additional option for transmissions is to perform "sub-transmissions". These can be used to indicate to the slave that either one portion, or one instance of a transmission has finished, but that the overall transmission is going on (used for example by the dense gate array, to receive multiple streams of the same state). A master indicates the end of a subtransmission by asserting `last` and `start` at the same time, during an ongoing transmission. Evidently, there is again no communication between master and slave about whether a transmission is going to comprise sub-transmissions. That information has to either be irrelevant, or be known upfront.

As a side note, there is a full specification for the bus in the delivered code documentation, which ensures well-defined operation in all cases, and contains cycle-accurate signal assertion rules. The level of detail in this discussion was restricted to necessary aspects for following this chapter.

Engine Data Bus - Design

One design goal for the engine data bus was simplicity at the engine side, as indicated. The element update formulas are a key factor in achieving that goal. They provide a level of independence between computations that in most cases eliminates any need for knowledge about the state itself, from the engine's perspective - as an example, for processing the element with label 11 during an X gate on state qubit 2, it is irrelevant if the state has 4 or 22 qubits. The only case when the engine requires state information from the memory back end is the tensor operation. This characteristic allows to almost

completely avoid counters in the entire engine, which saves hardware, and prevents crossties between data lanes (because the counter would need to check all lanes' valid/invalid fields to update).

Additionally, the lack of slave-side flow control means that the engine modules do not have to determine a dynamic ready-state across all lanes. This is appropriate because all computations in the engine have constant latency. Therefore, once the engine is ready, there is no way for the computation to stall, and thus no need for the engine to perform any "throttling". Naturally, engine modules could in several cases simply hard-wire the `ready` signal to be asserted, and allow logic optimization to simplify the control overhead. The only situation that would require input data throttling is if the data sink after the engine is not ready for new data. Since that sink and the data source are the same module, namely, quantum state memory, it was chosen to handle data sink stalls internally in that module - with the positive side effect of only having to implement the solution once, instead of potentially per individual engine module.

Lastly, the simpler the bus is, the easier it becomes to distribute it across the chip. The system contains multiple multiplexed engine modules, and possibly, on the other side, multiplexed physical memories. Therefore, data can have to travel relatively large distances across the chip, and have to be pipelined to meet timing targets. The absence of slave-side flow control (within transmissions) helps with employing simpler pipeline stages, because with a continuous (buffered) data handshake, pipelining stages must be equipped with data buffers, and require a small portion of additional logic. With a one-time ready signal, one can afford the (mostly hidden) additional latency of first letting the ready signal traverse the pipeline into one direction, before data transmission begins into the other direction.

Based on the above considerations, this section is concluded with an assessment on the importance of a custom data bus system: It brings advantages in data distribution and operation control, and it greatly helps with distributing responsibilities between engine and memory back end. However, the assumption is that a generic data bus like AXI Stream would have worked sufficiently well, because it supports a "transmission/packet" concept via the `tlast` signal, and because the on-chip bus topology appears still manageable.

5.5. Core Layer - Computation Engine

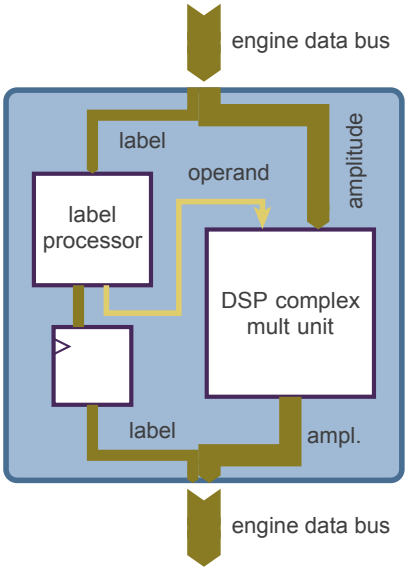
After the previous sections introduced global and recurring components of the FPGA implementation, this and the following section treat individual modules. First, this section focuses on the *engine* (see Figure 5.1). The discussion is analogous to the high-level perspective provided in Sections 4.3.1 to 4.3.4. In most cases, the translation to FPGA was achieved by employing a complex DSP multiplier, or a derivative thereof, and directly deriving the control signals for the output engine data bus from the control signals of the input engine data bus. Next to these, the realization of buffers and accumulation circuitry will be discussed.

5.5.1. Sparse-Gate Operation Module

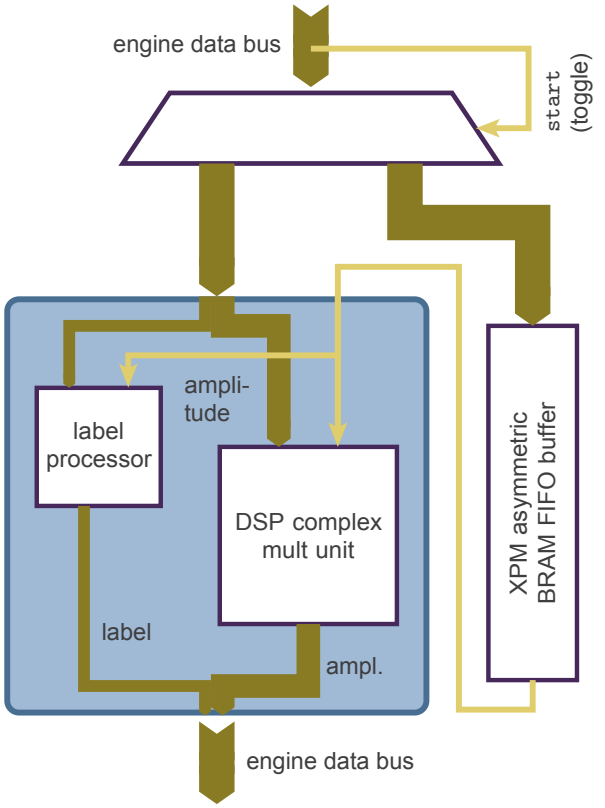
As has been the case in previous stages of this thesis, the sparse gate module represents a "trivial" case among the engine modules, but serves as a good example to introduce concepts that translate to the other units. The circuit is depicted in Figure 5.4a. In terms of data processing, the multiplier in this module is indeed exactly the complex DSP multiplier introduced in Section 5.4.1. Since the label processor is a combinatorial circuit, while the complex multiplier has multiple register stages, the label processor output (the new label k^*) is pipelined for latency adjustment, such that the throughput is not affected. Invalid input data elements are ensured to be forwarded to the output as invalid elements. Considering control signals, the input `ready` signal is always asserted, since there is no situation when this module is not ready to accept a transition. For the `start` and `last` signal of the output bus, it is sufficient to pipeline the respective input data bus signals through to the output, making sure that the pipeline depth matches the datapath latency. The computation does not entail anything that would alter the protocol flow of a transmission.

5.5.2. Tensor Operation Module

The predominant difference between the tensor module (depicted in Figure 5.4b) and any other module is that it operates on 2 input states, instead of 1. Still there is obviously only one input data bus, such that the module has to alternate between the two input states, and buffer the static state (named



(a) *Sparse-gate* (engine operation types *diagonal*, *reverse-diagonal*, and *swap*)



(b) *Tensor - state 2* (Figure 4.2b) is received in batches via the same data bus as *state 1*, prior to streaming *state 1*, into an XPM BRAM FIFO buffer.

Figure 5.4: FPGA implementations for the high level schemes of the *sparse-gate* and *tensor* operations. The label processors/checkers contain the element update formula circuits, and have access to any relevant operation parameters (such as target qubit, operation matrix, measurement type)

state 2, in accordance with Section 4.3.2 in the previous chapter). Buffering is done by one Xilinx Programming Macro (XPM) asymmetric BRAM FIFO. The buffer can be made fairly large, because the engine otherwise barely uses any BRAM, and because regardless of the data bus width, and hence number of parallel CUs, the module only needs one buffer. "Asynchronous" refers to the input and output width: The input side matches the data bus width, while the buffer output is only one vector element wide, since only one element is needed across the entire module during one iteration of the streamed state (state 1). An advantage of using Xilinx hardware is that the toolchain offers pre-existing macros for standard hardware structures, like this buffer.

From a computation standpoint, the module is relatively straightforward. Each unit contains a standard complex DSP multiplier, whose inputs are the current registered amplitude, and the data bus input. The label processor applies the "iteration offset" from Equation (3.17), followed by a pipeline stage for multiplier latency adjustment. In order to do so, the label processor requires the size of the streamed state 1, which is not an operation parameter. Instead, it gets that size from the quantum state memory module, upon loading the information from the memory allocation table. The core control module makes sure that the state size information has propagated to the engine tensor module in time (which can even happen while already filling up the state 2 buffer).

Protocol-wise, the tensor module showcases how a module can actively make use of the available databus control signals and the subtransmission feature, combined with mutual pre-existing information on semantics. The input and output protocol flow of a full tensor operation is illustrated in Figure 5.5. First, state 1 usually has to be streamed multiple times, while applying the buffered state 2 elements one-by-one. Subtransmissions are defined for exactly that purpose, so the module always expects as many subtransmissions of state 1 as there are elements in the buffers (it is up to the quantum state memory operation FSM to ensure this is the case). Secondly, the input `start` signal can be used for

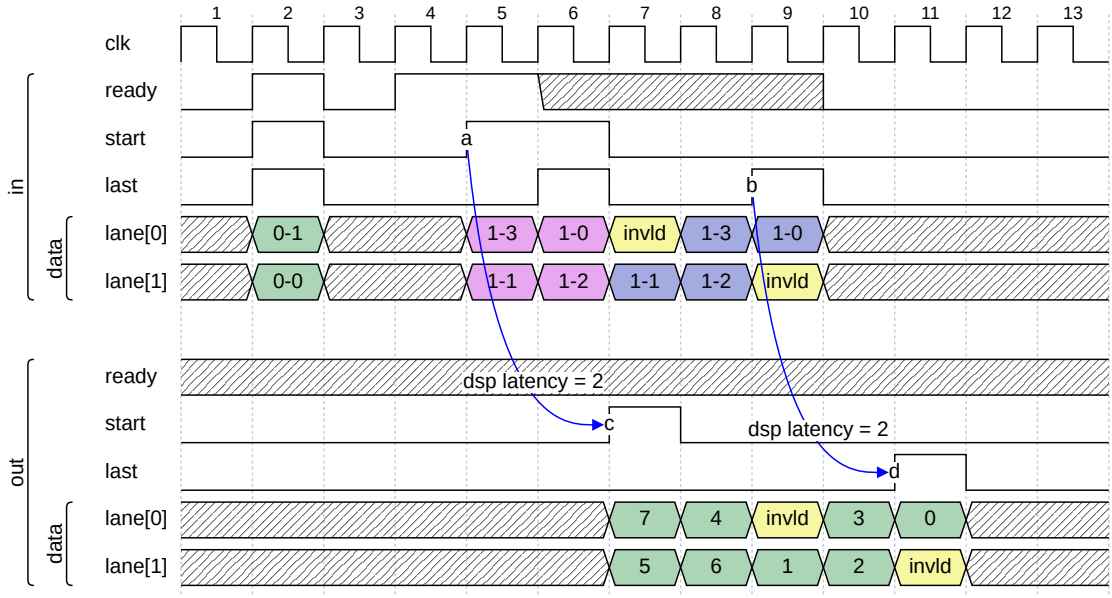


Figure 5.5: Engine tensor unit computation process (merging a 1-qubit/2-element and a 2-qubit/4-element input states). Data elements labeled `[<state>-]<element-label>`. 1. Register elements of first state (cycle 2). 2. Multiply single amplitudes of first state with all amplitudes of second state, and update labels accordingly (cycles 5-11, 2 subtransmissions of state 1, assuming a DSP latency of 2 cycles)

toggling the input data routing between the buffer and the CUs. The computing scheme demands that transmissions alternate between state 1 and state 2, while the first transmission is always a chunk of state 2 (going to the buffer), and the last transmission is always state 1. Therefore, it suffices for the tensor module to toggle between the two datapaths at every transmission start (not subtransmission!). Subtransmissions come in handy here, because they allow a clear distinction between just a new iteration of state 1 to process with the next buffered element of state 2 (no toggling), and new data for the elements buffer.

Operating the output bus correctly does require one (coarse) counter, deviating from the general engine design philosophy. The output bus `start` signal is trivial, it is the delayed `start` signal of the input bus, when idle (although it still takes time until the first “valid” data, because the buffer is filled first, but in this case there is no downside to having a long, mostly empty transmission). The output `last` signal is evidently a delayed input `last` signal, when finishing the last set of state 1 iterations. Just which iteration is the last one can only be determined from the size of state 2, and the buffer size. The buffer size is known to both engine and operation FSM, but the size of state 2 needs to be forwarded to the engine from the memory allocation table, alongside with the size of state 1. The tensor module then has to count the iterations, but at least not single elements.

5.5.3. Dense-Gate Operation Module

Clearly one of the more complex engine modules is the dense gate array, and by extension, its Processing Elements (PEs) (Figure 5.6). Most importantly, as much functionality as possible has been shifted into the DSPs, to use a maximum of their dedicated circuitry - which is present regardless whether one uses it or not - and save fabric logic resources. So far, only the output cascading was used, next to the standard multiplier-ALU datapath. The dense gate multipliers make use of the input section pipelining for cascading, and of dynamic output register feedback for accumulation. The entire DSP configuration for the first 2 levels in a cascade is depicted in Figure 5.7. The following paragraphs elaborate on the mentioned functionalities.

Computation

Both the amplitudes and the labels need to be cascaded by the array elements. The labels are processed in fabric logic, and therefore also cascaded in fabric logic. The amplitude can make use of the DSP column routing. The first unit in a cascade/column gets the amplitude as the A input. From there on,

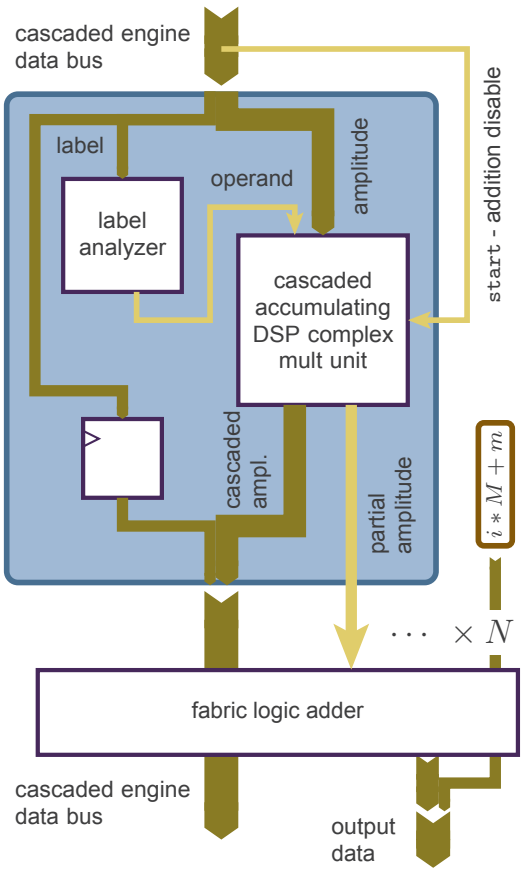


Figure 5.6: FPGA implementation of the high level scheme (Figure 4.2) for the *dense-gate* element update formula. The *label analyzer* contains the element update formula circuit, and has access to any relevant operation parameters (such as target qubit, operation matrix). The DSP multiply unit (Figure 5.7) uses the DSP's built-in accumulation circuit, which reduces the external accumulation module from Figure 4.3 to an adder. The *enable* signal for the multiplier/DSP (Figure 4.3) is hence split up: "Ignoring" a label is implemented as 'operand $= 0$', disabling/resetting the accumulation at a new iteration is directly derived from the (pipelined) *engine data bus* `start` signal

it propagates through the whole array depth M via the dedicated ACOUT/ACIN routes (provided that M does not exceed the physical column depth). The latency is matched between the amplitude and the label cascading. A downside is that, since adjacent DSPs thereby belong to different levels, one can no longer couple pairs of adjacent DSPs for collectively computing the result's real or imaginary component - in contrast to the standard complex DSP multiplier. The addition step now needs to be done in fabric logic, which still allows for good on-chip locality, if one uses adjacent DSP columns in the chip in pairs, for real and imaginary components. Therefore, 1 array "lane" uses 4 DSP columns.

In order to perform accumulation, the DSPs allow to "feed back" the P result register to the ALU via either the W or the Z multiplexer, for addition with the next multiplier result for instance. It is useful to use the Z multiplexer in this case, instead of W, because the ALU only supports specific combinations of adding or subtracting the multiplexer outputs - among which both $(\mathrm{Z} + (\mathrm{X} + \mathrm{Y}))$ and $(\mathrm{Z} - (\mathrm{X} + \mathrm{Y}))$, but not $(\mathrm{W} - (\mathrm{X} + \mathrm{Y}))$. With the Z multiplexer, the DSP can therefore still compensate for $i \cdot i = -1$, which with the W multiplexer would require a fabric logic adder. In either case, "resetting" the accumulation between iterations of state 1 is done by setting multiplexer Z output to 0 via the OPMODE signal for 1 cycle, which in essence just disables the feedback path. Since the accumulation part is thereby taken care of, there is only a level-wide fabric logic adder, for what in the high level design was introduced as level-wide accumulation. A combinatorial fabric logic multi-operand addition was sufficient for the prototype, because that is not implemented yet with full theoretical data bus width, or full clock frequency.

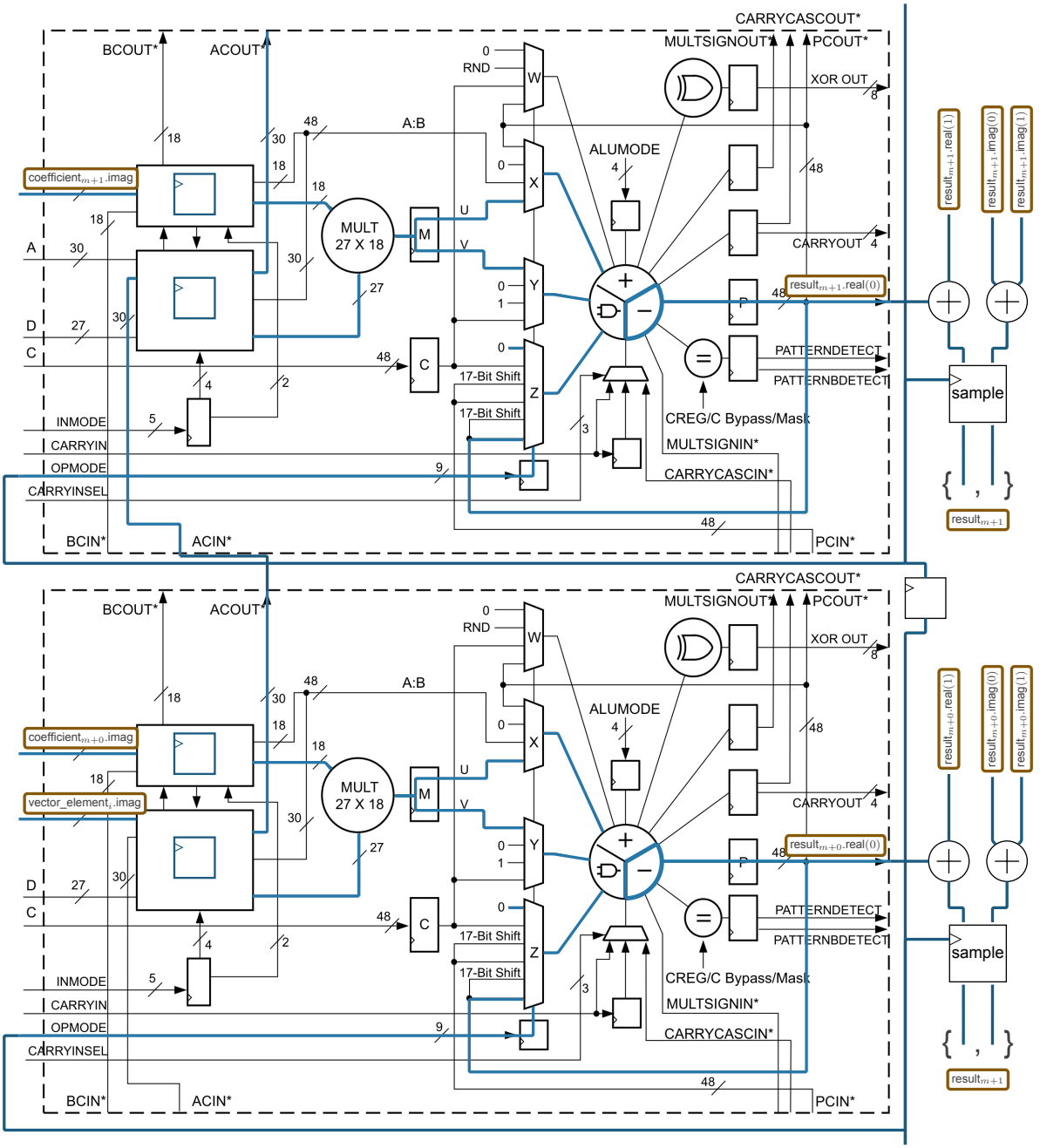


Figure 5.7: first two elements of the "imaginary $\times$ imaginary" cascade in the dense-gate engine module computation array; indicated on the right-hand side the 3 parallel cascades (second portion of *result.real*, and both portions of *result.imag*); 4 cascades form 1 lane, results are accumulated across all lanes. The ALU feedback/accumulation path (via multiplexer Z) can be switched off across all cascades and lanes synchronously, to reset the P output register after a (sub)transmission.

Databus Protocol Operation

In the same way as in the tensor module, the input databus control signals help to orchestrate the operation within the module. Figure 5.8 shows the protocol flow for one full operation. The input, analogous to the streamed state in the tensor module, consists of an appropriate number of subtransmissions of the full input state, with the number depending on the array depth M. The output `start` is again the delayed input transmission start. The input `last` signal (hence, the end of any subtransmission) is used for disabling/resetting the DSP accumulation. It is delayed to match the DSP datapath latency, and then pipelined through the array levels. At each array level, it turns off the DSP feedback path for one cycle, and samples the current P result register to a module output register, from where it is written to the

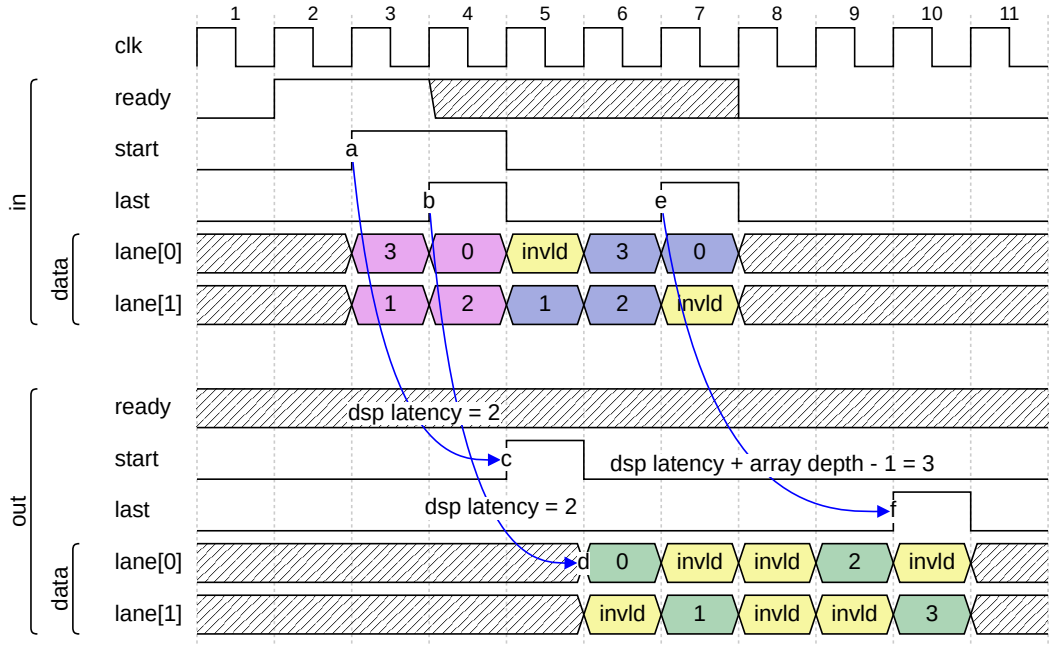


Figure 5.8: Data bus input to output for the engine dense gate computation module. Data element numbers represent vector indices. Figure for a (theoretical) DSP (figure 5.7) latency of 2 cycles, a 2-level deep DSP array, and a 4-element/2-qubit calculation. 2 input subtransmissions (4 elements / 2 levels), 1 output transmission. Note that the output vector elements are always in-order, regardless of the input.

output data bus - alongside with the current label $(i \cdot M + m)$. This way, the array writes to the output bus level-by-level, which allows the later levels to finish computation of one subtransmission, while the earlier levels are already ready to receive data for the next subtransmission. The output transmission is terminated by delaying an input transmission finish signal (not subtransmission) in the same way, and propagating it to the output `last` signal once it has reached the last array level.

5.5.4. Measurement Operation Module

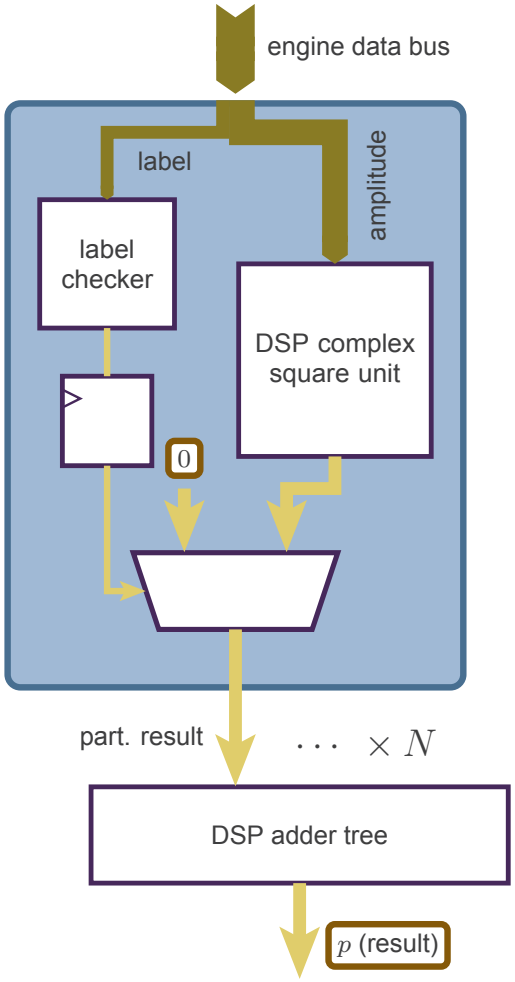
This section is subdivided such that it first discusses the parallel PEs of the measurement module, and then goes into detail on specific submodules. In terms of PEs, there are the measurement probability phase (phase 1), and the post-measurement state phase (phase 3). Afterwards, there will be a specific discussion of the entropy source, which is used to determine the measurement outcome, and on the implementation of the Goldschmidt-Algorithm for inverse square-root computation.

The measurement expectation value CU (Figure 5.9a) structurally consists of components that are already known, at this point. There is a complex DSP multiplier, which in one DSP squares the data input's imaginary component, and in the other DSP squares the real component (performing Equation (4.2)) - with the "result" ALU configured to perform the necessary subtraction.

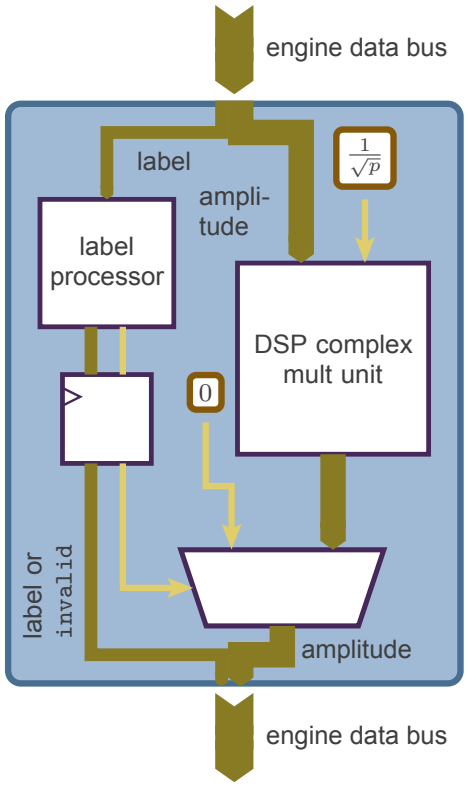
Since the PE results need to be accumulated across the entire parallel array, and for the entire input state, there is a subsequent accumulation structure. In contrast to the previously introduced dense gate module, the measurement module already employs a scalable multi-operand accumulation method, in the form of a DSP-based accumulation tree. The tree nodes are formed by pairs of DSPs, which are each wired to perform 4-operand (48-bit) addition (schematic in Figure 5.10), and bypass the multiplier. The root node has a switchable DSP result feedback path enabled, analog to the dense gate multipliers, which allows the tree to be used for accumulation (with reset).

Once the measurement probability module encounters the input transmission's `last` bit, it delays it for resetting the accumulation, and in the same cycle comparing the adder tree's result value to the current Random Number Generator (RNG) value (see below), for obtaining the measurement result.

The post-measurement state module (Figure 5.9b) does not require much explanation. It is essentially the high level design circuit, with the standard complex DSP multiplier. The label processor determines both the label update, with respect to destructive or non-destructive measurement, and multiplexes the



(a) *Measurement* probability (phase 1). The *enable* signal (Figure 4.4) is implemented as an output multiplexer. The adder module is a DSP-based 48-bit adder tree (Figure 5.10).



(b) *Measurement* post-measurement state (phase 2) - input elements that do not result in an output element (because of destructive measurement) are streamed out as `invalid` elements.

Figure 5.9: FPGA implementations of the high level schemes (Figure 4.2) of the *measure* element update formulas. The label processors/checkers contain the element update formula circuits, and have access to any relevant operation parameters (such as target qubit, operation matrix, measurement type).

multiplier output, to set amplitudes to 0 that have been "measured out" in a non-destructive measurement. The output data bus protocol flow follows the input, subject to the according PE latency adjustment.

Entropy Source

The criteria for choosing an entropy source were: a small hardware footprint, uniform distribution, and a "reasonably good level of randomness". These criteria point towards a PRNG, because a True Random Number Generator (TRNG) means unnecessary complexity, in a simulator implementation which is not security-critical.

LFSRs fulfill the formulated requirements. These are shift registers, with specific bits ("taps") XNOR'ed and fed back to the input, which makes them simple to realize in hardware. An example for a 4-bit LFSR is shown in Figure 5.11. If tapped at the correct register stages, an n-bit LFSR can go through almost all 2^n possible register states (except for all 1's, without additional circuitry) before repeating, and thus reach period lengths of essentially 2^n. The correct taps for maximum period length (and minimum amount of taps) can be found in the literature, for example in Xilinx application note 052 [39]. With the taps guaranteed to cycle through all possible numbers, the number distribution over one period is perfectly uniform (neglecting the missing all 1's state). Given how simple it is to achieve long period times, for many experiments, the distribution was considered to offer "good randomness" (for instance, a 48-bit LFSR at $1\,\mathrm{GHz}$ has a period time of over 78 hours). It was not investigated to which

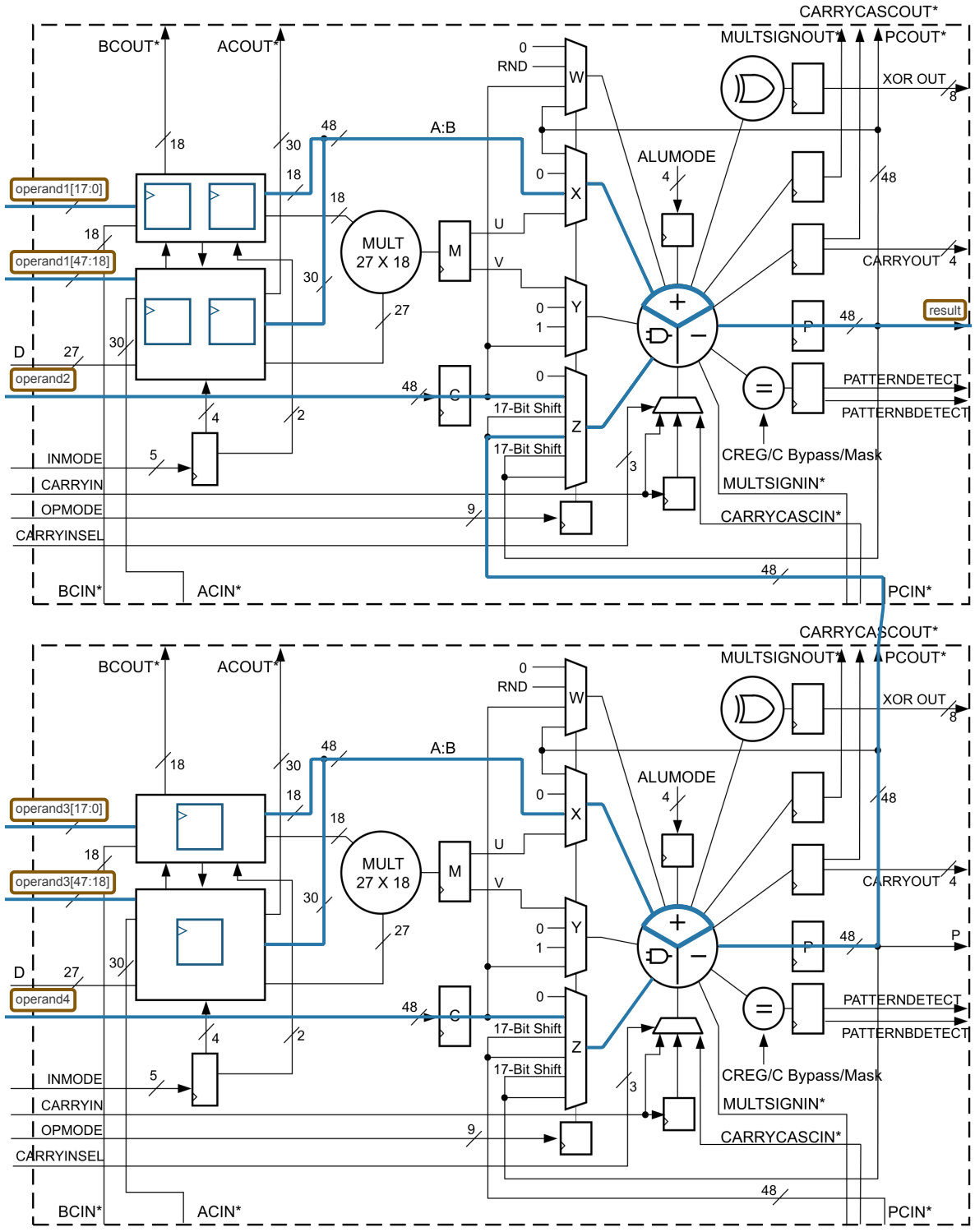


Figure 5.10: DSP 4-operand 48-bit real number addition node, used in accumulation tree for combining partial results from multiple lanes of DSP measurement expectation value calculators (Figure 5.9a)

extent LFSRs might have a “dynamically shifting bias” across their period length. However, the empiric measurement probability results in Section 6.2.3 show no indication that the system is affected by a such effect to a problematic extent.

LFSRs can either be used as pseudo-random bit generators, if one only looks at the last bit, or as counters in pseudo-random order, if one looks at all the bits. This implementation uses the latter method,

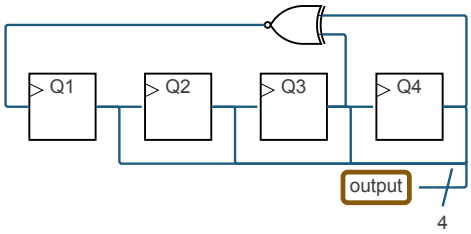


Figure 5.11: Example 4-bit Linear Feedback Shift-Register, with all register outputs used to form a pseudo-random 4-bit number, like in the measurement unit entropy source (which uses 48 registers/bits). Feedback taps for maximum period time. Registers are initialized to all-0s, to prevent the lock-up all-1s state.

where a 48-bit LFSR is compared to the 48-bit full length measurement probability, computed by the measurement probability module's DSPs. If the random value is larger or equal, the measurement outcome is 1, otherwise, the outcome is 0. "Larger or equal" is slightly beneficial here over "strictly larger", because the random value can not become all 1's, while all 0's is possible. Therefore, there is an extremely subtle bias, which "larger or equal" counteracts. It was not studied to which point a smaller LFSR would have given equally satisfactory results, or if a larger bitwidth would have given better results.

Goldschmidt-Algorithm on DSP48E2

As illustrated in Figure 4.5, the Goldschmidt-Algorithm, necessary to compute the normalization factor for post-measurement state computation, can be represented by a 3-multiplier hardware structure with a feedback line. The circuit structure translates flawlessly to 3 DSPs, as shown in Figures 5.12 to 5.13. There are however 2 questions to be answered, for a full FPGA implementation: First, not all involved variables, including the result itself, fall into the number range $[-1, 1]$, for which the system-wide standard fixed point number representation was defined. One needs to therefore determine the best-performing distribution between integer and decimal bits for every variable. In the same context, it may have an impact how one maps variables between the $18\,\mathrm{bit}$ and the $27\,\mathrm{bit}$ port of the multipliers, in other words, "which variable can best afford a reduced precision". The second remaining task is to design a viable scheme for determining an initial value x_{init}.

Looking at the initial value x_{init}, the (theoretical) ideal starting point would be $x_{\text{init}} = \sqrt{x}$. However, if there was a feasible way to do that, the entire algorithm would be obsolete. Therefore, one needs an approximation to $\sqrt{x}$, that is manageable to perform quickly in hardware. For this thesis, the below equation was developed, which is explained in the following paragraphs:

$$\text{with} \quad \text{shift}(x) := \begin{cases} \left\lfloor \frac{\lfloor \lfloor \log_2(x) \rfloor \rfloor}{2} \right\rfloor, & \text{if } x < 1 \\ -\left\lceil \frac{\lfloor \lfloor \log_2(x) \rfloor \rfloor}{2} \right\rceil, & \text{if } x \geq 1 \end{cases}$$
$$x_{\text{init}} = x \cdot 2^{\text{shift}(x)} \tag{5.1}$$

The thought process is as follows: Computing the exact square root of an integer (or fixed-point number) in binary is doable, **if** the number is an even power of 2. In this case, the binary representation has exactly 1 bit set, which is an even number of bits "away" from the decimal point (1-indexed counting). In these cases, the square root of the number is computed by "halving the distance between the set bit and the decimal point". As an example, $16 = 2^4$ has only bit 4 set (1-indexed). Halving the distance between the set bit and the decimal point would set it to bit 2, which yields $4 = \sqrt{16}$. The same concept works for numbers $0 < x < 1$, where the bit is still shifted "towards the decimal point", just that now means shifting it towards more significant bits. The idea behind Equation (5.1) is to round numbers to the next even power of 2, rounding in the direction of the decimal point, and then "halve the distance of the remaining bit, to the decimal point". As the following paragraph will show, that is still not completely trivial in FPGA hardware, but feasible within 1-2 clock cycles.

The hardware implementation of Equation (5.1) is explained only for numbers $0 \leq x \leq 1$, because x is a quantum state vector amplitude, which can not exceed 1. This only means that the explanation can be restricted to "one side" of the decimal point, but does not take away from that it would work for integer numbers as well. The resulting implementation consists of these steps:

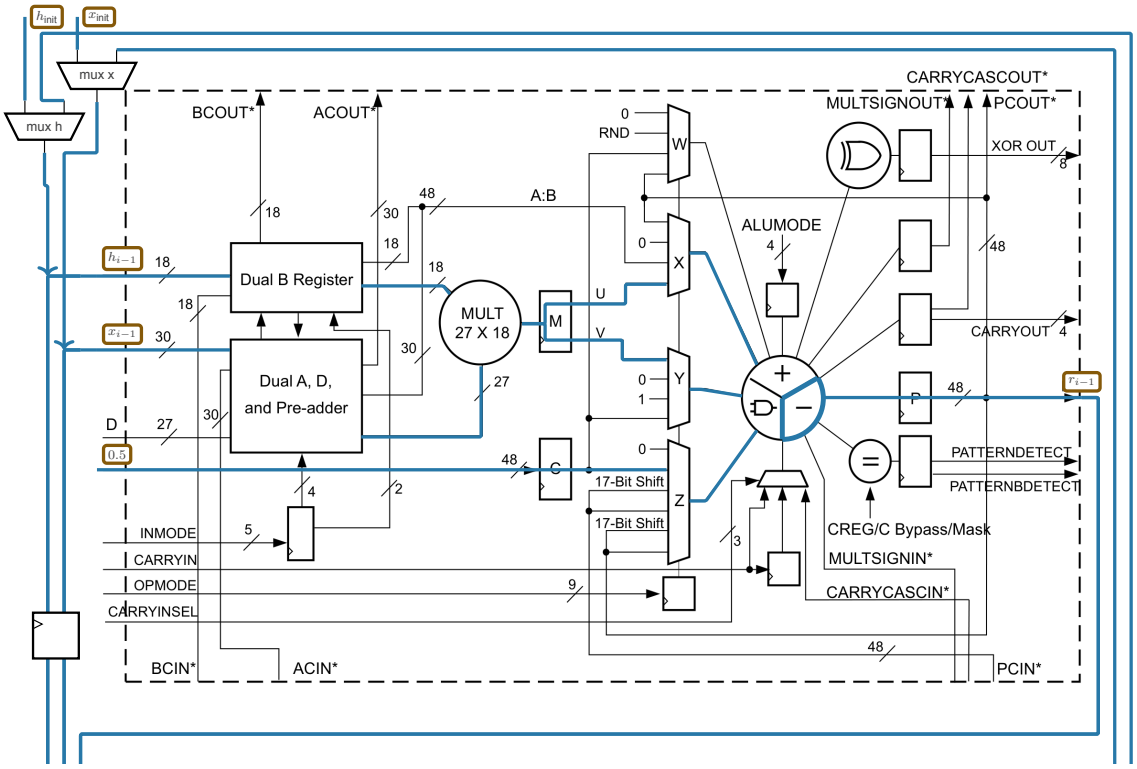


Figure 5.12: [first/upper half] DSP48E2 implementation of the Goldschmidt algorithm multiply-addition scheme depicted in figure 4.5

1. If the integer bit is set, assume $x = 1$. In that case, $\sqrt{x} = 1$, therefore $x_{\text{init}} = 1$. Return and skip all remaining steps.
2. If $x = 0$, $x_{\text{init}} = 0$. Return and skip all remaining steps.
3. Find the index (from the decimal point) of the highest fractional bit set.

   Note that this step is a “find-first-set” operation, which is problematic for combinatorial circuits in terms of clock frequency, and only tolerable for small enough operands.
4. Right-shift (logical) the determined index by 1.

   Note how the previous step only rounded the number to an arbitrary power of 2, but the right-shift in this step automatically turns that into rounding to an even power of 2. (example: $x = 0.125 = 2^{-3}$, distance to decimal point is 3, 3 right-shifted by 1 bit is 1, which is the same result as if x had been $2^{-2} = 0.25$)
5. left-shift (logical) x by the result of the previous computation, to gain x_{init}.

The corresponding hardware circuit is depicted in Figure 5.14 (with the find-first-set as one functional block, because it was implemented as a function in the hardware code, and then inferred by the synthesis tool). While this circuit is not “particularly efficient” in hardware - due to the find-first-set - it only takes 1 cycle to compute, and it did not pose a critical path to the prototype implementation. Additionally, the estimate seems to be “good enough”, since the algorithm performed with good accuracy through simulations and hardware testing (see Section 6.4.1).

The number representations for intermediary variables and the result, as well as the assignment of multiplier ports, were mostly determined empirically. The process is documented in Section 6.4.1. The only theoretical aspect, which is discussed here, consisted of “categorizing” numbers by their range. This led to defining two classes of numbers “in the same order of magnitude” (from a power of 10 standpoint), which would each get their own number representation, to narrow down the design space: x is derived from the input, therefore considered “in the same range” as a state vector amplitude. The fixpoint of the algorithm is $r = 0$, which makes it a number close to 0 (provided a good enough x_{init}). Therefore, for x and r, the same number representation is used as for vector amplitudes (1 sign bit, 1 integer bit, 25

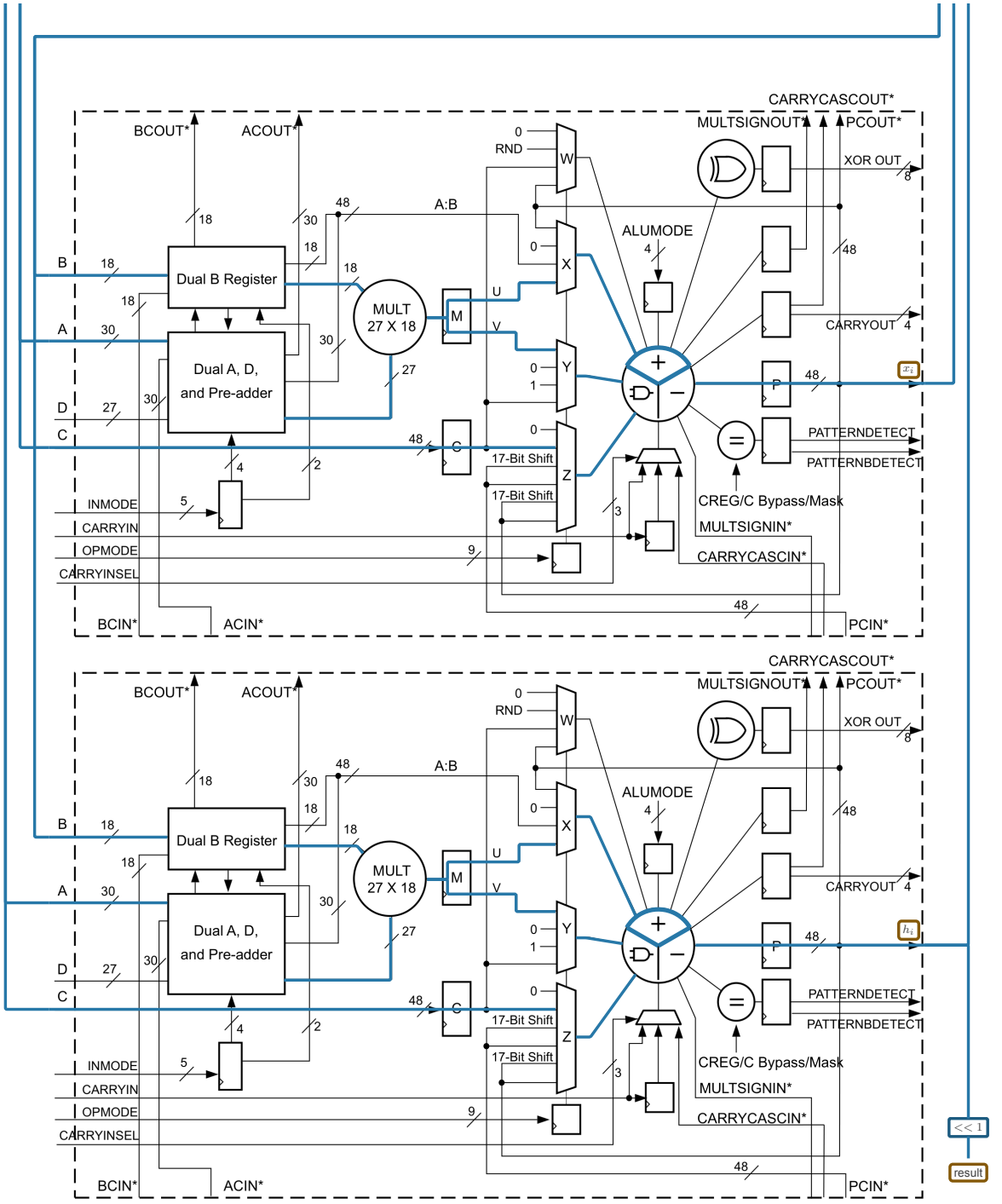


Figure 5.13: [second/lower half] DSP48E2 implementation of the Goldschmidt algorithm multiply-addition scheme depicted in figure 4.5

fractional bits). Theoretically, the sign bit is not necessary, because there are no negative inputs in this application. Practically, it can not be eliminated, because the DSPs expect two's complement inputs. h is considered to be "in the order" of the result, because the result is $(h \cdot 2)$. Therefore, these get the same number representation as well. The best-performing representation turned out to be **1 sign bit, 13 integer bits, 13 fractional bits**. As mentioned at the beginning of the paragraph, Section 6.4.1 elaborates on how this value was empirically obtained in simulation.

Assigning variables to wide and narrow DSP ports can be done per individual DSP, and per port. Since this is a prototype, the experiments were restricted to the first-stage DSP's inputs (x and h). r

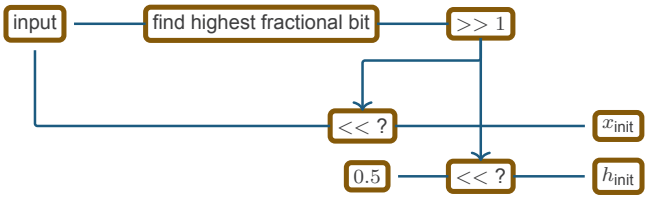


Figure 5.14: Implementation of the FPGA-tailored Goldschmidt algorithm initialization circuit for $0 <$ input < 1, trivial cases input $\in \{0, 1\}$ are omitted (see Equation (5.1))

was connected to the narrow port of the second-stage DSPs, with good enough results to not require any immediate change. A future simulation can quickly show if that was the optimal decision to take. In simulations (also elaborated on in Section 6.4.1), making h the wide port, and x the narrow port of the first-stage multiplier (as it is shown in Figure 5.12), was found to produce slightly more accurate results than the other way around.

The algorithm was also found to perform best on the inspected input number range with 5 iterations (and 4 iterations performing significantly worse). In the current configuration, the inverse square-root implementation consumes 3 DSPs, and takes 22 clock cycles to compute.

5.6. Core Layer - Quantum State Memory

Considering the quantum state memory module, the high-level discussion in Section 4.4 focused on the module interaction. This section does the opposite, in that it discusses in detail the separate modules of the "execution block" (Figure 4.7). The first portion concerns memory management, thus quantum state date in memory, while second portion treats quantum state data movement. The "control block" is not explicitly elaborated in this section, because there are no implementation-dependent details beyond the state machines that were presented in Section 4.4.5.

5.6.1. Execution Block - Memory Management

Physical Alveo U55C Quantum State Memory

For quantum state memory, the delivered prototype exclusively uses the U55C's on-chip HBM, and no local on-chip memory, because HBM is the predominant target memory for the accelerator scheme. Since HBM is a multi-channel memory (8 memory controllers/16 channels per stack, 2 stacks), there are degrees of freedom in which, and how many channels are used for reading and writing respectively. In that regard, the prototype is designed to keep implementation complexity at a minimum. Therefore, only two ports are used - one hard-wired for reading, one for writing - and the hard-IP crossbars are activated to allow both ports to access the full memory, at the cost of additional access latency. Furthermore, the available 2 HBM stacks are (logically) split: One $8\,\mathrm{GB}$ stack is used for quantum state memory, the other one (that is physically closer to the FPGA's PCIe transceivers) acts as shared memory for data exchange with the host system, as shown in Figure 5.15. Therefore, architecturally, the second memory stack is part of the intermediary layer, not of the core layer.

Dedicating half of the available on-chip main memory to data exchange with the host system may seem wasteful, and it likely is. It is the least-effort solution for a proof-of-concept prototype, but is considered appropriate for the prototype, since there are low-overhead ways for future design iterations to eliminate that flaw. First, several Alveo cards, among which the U55C, allow for shared accelerator memory to be allocated in host system memory, instead of on-board memory. In that case, all that needs to be changed is a parameter in the Vitis toolchain kernel configuration file. Another solution is utilizing a significantly smaller shared memory as a "buffer", such that state vectors are exchanged in batches, instead of in full. Again, the changes to the accelerator are marginal. Instead of one large AXI transaction, one would issue multiple smaller transactions, and introduce a synchronization mechanism with the host software side.

Quantum States in Memory

Section 5.3 explained that quantum state elements, in the prototype, are $83\,\mathrm{bit}$ data structures (2 $27\,\mathrm{bit}$ amplitudes, one $29\,\mathrm{bit}$ label). As a design decision, these are aligned in memory to $128\,\mathrm{bit}$ boundaries. While that reduces the effective memory space by roughly $35\,\%$, the additional complexity of handling

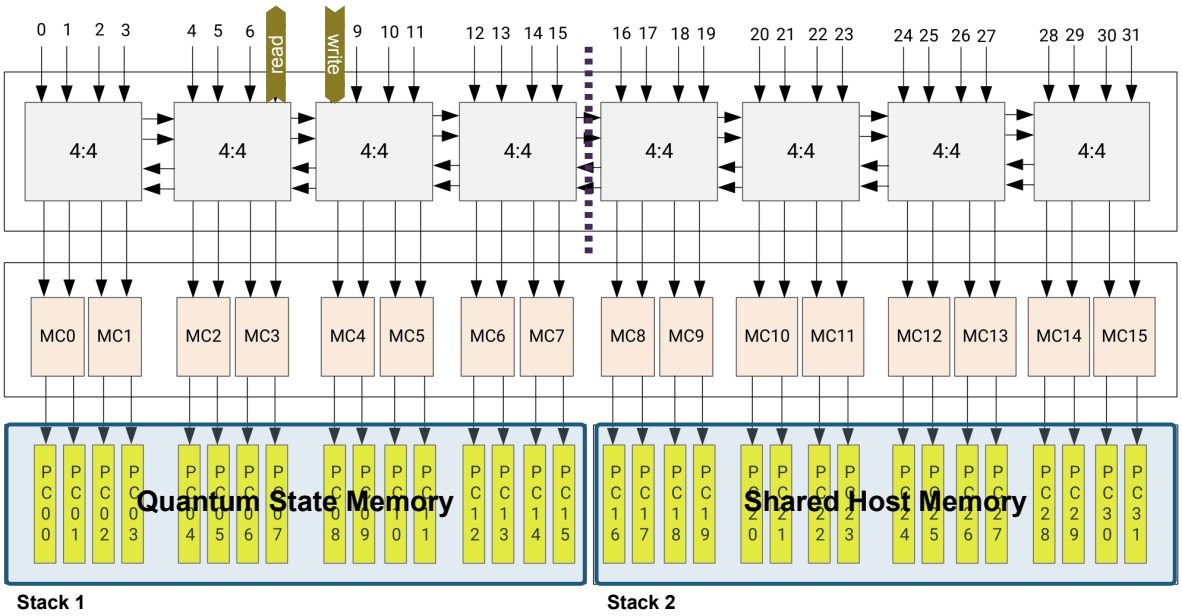


Figure 5.15: HBM overview [4] with prototype-specific partitioning: Stack 1 for Quantum State Memory, accessible by the kernel only, access via 2 AXI4 buses (one used as read-only, one as write-only). Stack 2 for state vector exchange with host application. Connectivity set up in Vitis platform compiler, prevents illegal accesses.

"non-padded" elements in memory is undesirable, at least for a prototype. First, aligned elements extremely help with memory address computation, because the number of elements per memory dataword is a constant power of 2, and because the size of a state vector is always a power of 2. The combination of these two makes it drastically more simple to determine the memory space occupied by a quantum state, compared to non-aligned elements. Dataword counters in the memory back end also need less bits, if they have guaranteed step sizes of (multiples of) a certain power of 2. Lastly, bit assignment between AXI memory words and engine data bus lanes can be hard-wired, with aligned state vector elements. Otherwise, it would always be a dynamic bitshift with 83 theoretical options (every bit position would have as many potential source bits as there are possible alignments in a memory word, which is equal to the bitwidth of a state vector element).

Another aspect to consider is the connection between how a state is fragmented in memory, and the achievable bandwidth when reading or writing that state. Differences here come from the fact that the memory at hand is composed of physically separate "pseudo-channels" ("PC" in Figure 5.15). Although two PCs share one physical memory controller ("MC" in Figure 5.15), the controller can operate both pseudo-channels at maximum throughput. It is logical that reading a state that resides in only one pseudo-channel, takes twice as long as if the same state was allocated "wider", across two pseudo-channels, which can operate in parallel. In a hypothetical scheme that still separates read and write ports (because a standard operation still needs to read and write at the same throughput), and that makes the entire 32-channel HBM accessible for quantum state memory, there can hence be a throughput difference of up to factor 16, depending on how input and output state are laid out in memory. The layout of quantum states in physical memory is henceforth called *state allocation policy*. Despite considerable optimization potential, the prototype uses the least complicated method, in assigning contiguous address regions to states (which in default address bit mapping fills up pseudo-channels first), like a classical array in a conventional computing system. A generally more efficient variant for a future implementation is presented in Section 7.2.2.

Memory Allocation Policy

As the previous paragraph motivated, the way that memory accesses are physically distributed across HBM memory can have a great impact on the resulting memory performance. To an extent, that does not only apply to how one single state is physically allocated, but also to the relation between source and destination memory during one operation. In the typical situation, where a pre-operation state is read, while the post-operation state is written, one can imagine that it benefits efficiency to read via

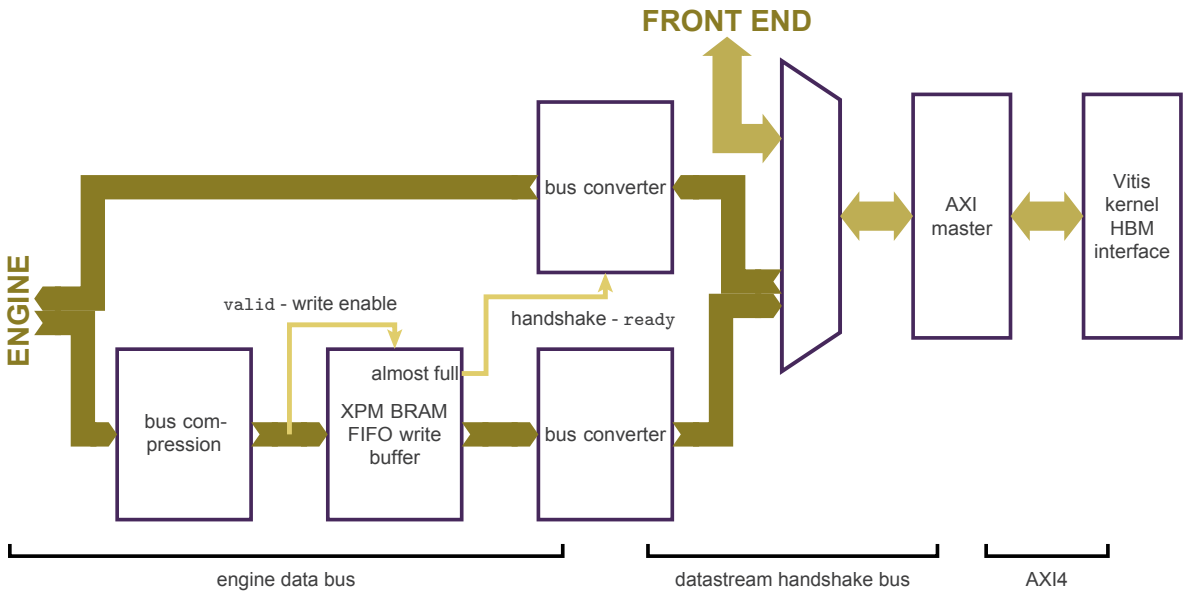


Figure 5.16: Quantum State Memory data multiplexing and processing. The Vitis compiler toolchain provides AXI4 interfaces to the enabled HBM interfaces (see Figure 5.15 “Quantum State Memory”). Custom AXI masters operate the bus, and convert to a simple point-to-point *datastream handshake bus*. Data is multiplexed to the *front-end interface* and to the *engine* (Figure 4.1). The *engine* datapath needs conversion to the *engine data bus*, and data loss protection (*bus compression* and *FIFO write buffer*, see Section 5.6.2).

one of HBM’s memory controllers, and write via a different one. The unit that decides about these allocations is the memory allocator. A second way in which the allocator can be affected is the state allocation policy, which can make it easier or harder for the allocator to determine whether or not two allocations collide. When finding a new allocation, an allocator algorithm or FSM will always, in one or the other way, have to check for a given candidate allocation if it conflicts with an existing allocation. For contiguous memory address allocations, that is relatively easy to check, but can become more complicated with other schemes.

Again, as was the case with memory channel accessing, the prototype opts for a relatively simplistic *memory allocation policy*: Find-first-fit. Upon request, the allocator starts at address 0, and continues iterating through the memory allocation table, increasing the start address, until it found a candidate allocation that does not collide with any occupied memory. The scheme in algorithm form can be found in the appendix, Section D.1. As an additional constraint, in the given allocation, the allocator aligns states to $4\,\mathrm{kB}$ boundaries, for seamless interaction with AXI4’s maximum transmission size of $4\,\mathrm{kB}$. Additionally, it needs to be noted that, in the given implementation, the computation time increases with the number of possible entries in the memory allocation table, and thus with the maximum number of quantum states in the system.

5.6.2. Execution Block - Data Back End

As explained in Section 4.4.1 (and depicted in Figure 4.7), the quantum state memory module has two dual-simplex data connections - one to the engine, and one to the intermediary layer. On the other side, it has AXI connections to the physical memory. That architecture comes with a series of tasks that the data back end needs to fulfill. On the memory end, there need to be bus masters to operate the AXI connections. On the accelerator end, besides multiplexing data, there needs to be suitable conversion between memory and engine bus protocols. Additionally, as was explained in Section 5.4.2, the module needs to handle potential stalls in writing back data coming from the engine during an operation, because the engine data bus has no slave-side flow control. The following paragraphs explain how the data back end accomplishes those functionalities. An overview of the architecture is given in Figure 5.16.

AXI Masters

Access to the main memory is provided via 512-bit wide AXI4 interfaces, generated by the Vitis compiler. From the accelerator hardware side, these are operated by custom-written AXI3/4 masters. The AXI

masters have a handshake data streaming interface on the “user” side, called *datastream handshake bus* in the following. It is almost identical to AXI Stream, and is therefore not explained further.

The AXI masters are written to support high clock frequencies, and arbitrary-length transactions. The clock speed performance is achieved by employing a reasonable amount of pipelining, keeping the module half-duplex, and by reducing counters to a minimum. Arbitrary transaction length is achieved by automatically splitting up an input transmission into as many AXI transmissions as are needed (AXI4 has a limit of $4\,\mathrm{kB}$ per transmission, regardless of length and burst size). Since the AXI masters are half-duplex, there are two instances in the prototype’s data back end, one for reading, one for writing.

Engine Data - Protocol Conversion

The data back end’s task of supplying all involved parties with data includes necessary databus conversions. No action is necessary for data exchange with the intermediary layer, because that path ultimately connects to additional instances of the same AXI masters than the ones that operate the on-chip memory. Conversion however is necessary between *datastream handshake bus* and *engine data bus*, for data involved in quantum state operations. The conversion is not completely trivial, because the datastream handshake bus has a handshake-based flow control, but no notion of packets, while this is the opposite for the engine data bus. The converters are explained in full in the appendix, Section D.2, including wave diagrams. The only characteristic that requires a mention here concerns the asymmetry between the two protocols in terms of flow control: Since the engine data bus has no slave-side “throttling” mechanism, there is no way for the converter in the “return path” (from engine to memory) to obey the datastream handshake bus `ready` signal during a transmission. The below paragraph on *Buffering & Throttling* explains how the system deals with that problem.

Engine Return Data - Compression

The engine databus allows for individual “invalidated” fields in data lanes, during a transmission. This ultimately causes a problem when writing back data to the memory (regardless of bus conversion, in this case): The design constraint is for the memory to operate using continuous writes and reads. Therefore, the datastream going to the memory can not have “mixed” datawords, that contain both valid and invalid elements. One thus needs to “filter out” invalid elements. In this paragraph, this process is called *bus compression*, because it compresses the possibly scattered valid data elements into cycles with only valid elements.

It is absolutely imperative to employ bus filtering, although that might not be self-evident. Obviously, the situation of “non-compressed” transmission cycles can arise from interfacing with a multi-channel memory (with non-deterministic access times), like HBM - where in a cycle, one channel might deliver data, while another one is busy loading a page. However, regardless of the memory interface, destructive measurements are almost **guaranteed** to cause non-compressed transmission, because half of the elements of the input transmission are “invalidated”, when computing the post-measurement state. Therefore, any implementation must have a means to filter out invalid data fields, before writing data to memory.

The problem seems to have a simple solution, from a software perspective, which however is infeasible in a hardware implementation. That intuitive algorithm could be loosely formulated as follows: “In every cycle, identify all data lanes that hold a valid data element. Write the valid elements to a FIFO buffer. As soon as the buffer is filled up enough to supply an all-valid cycle, write one full bus dataword from the FIFO to the output bus. While the buffer is not filled up sufficiently, keep the output dataword all-invalid”.

Two things in this algorithm, in this form, are either not doable in hardware, or scale terribly with the databus width: First, the algorithm must not hinder the data throughput, but a FIFO only accepts one new entry per clock cycle. Therefore, the buffer would have to be a set of random access registers instead, which becomes a roadblock when looking at the second flaw. “Identify a data field with valid data” results in a find-first-set operation, that on its own can already cause complexity issues in hardware. The eventual problem is that multiple of these are necessary, because the algorithm mandates identifying “**all** data lanes with valid data”, and that on top of that, all these instances are operating hierarchically (in order to write an element to the correct buffer position, a circuit needs to know how many valid elements were found before). A scaling number of hierarchically nested find-first-set circuits is impossible to implement in digital logic, at any reasonable clock frequency or throughput.

The solution to that problem is “stretching out” the above algorithm in space (resources) and time (latency), thereby preserving the necessary throughput. Figure 5.17 illustrates the implementation that

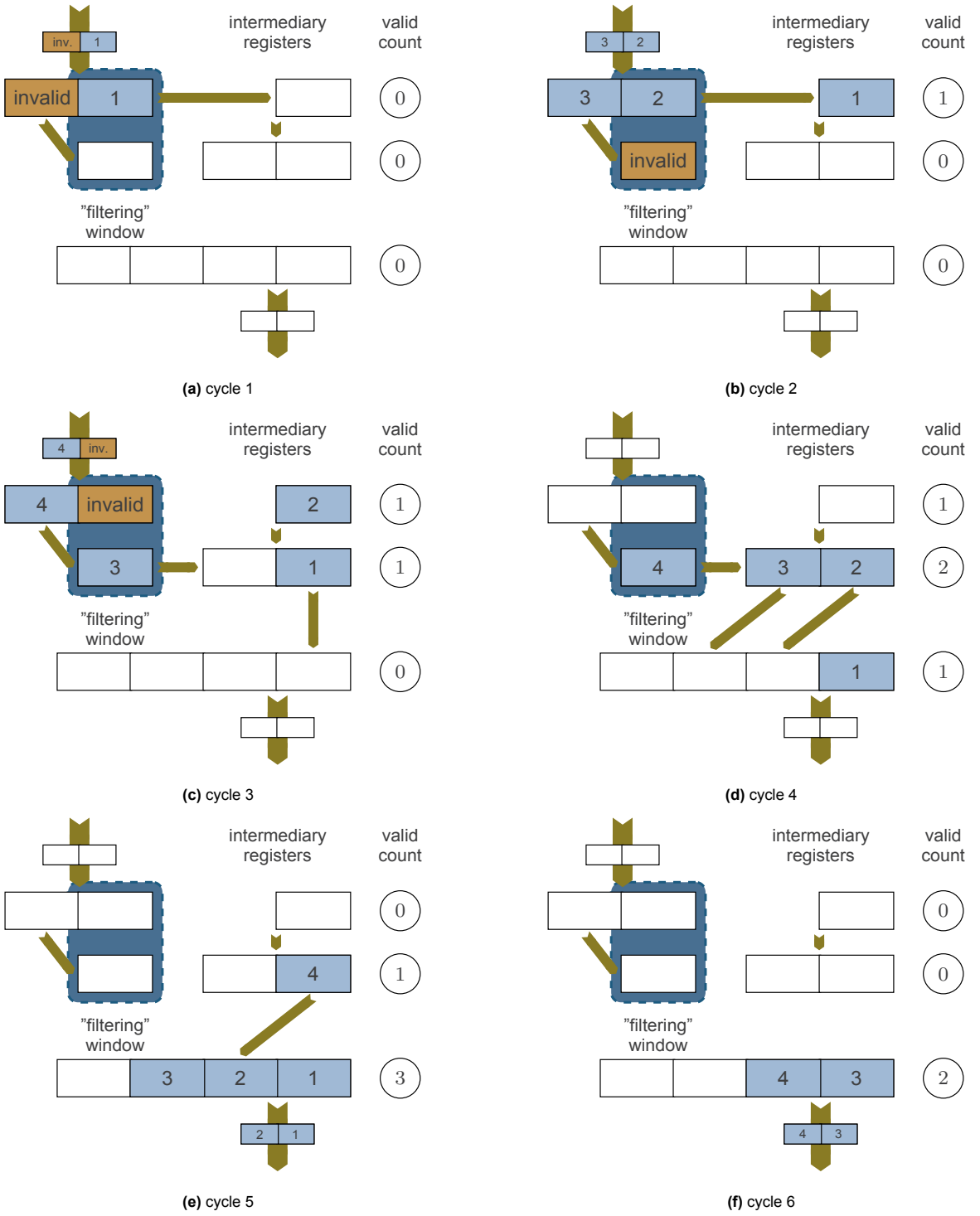


Figure 5.17: Dataflow of *engine data bus* compression/filter module, for 2-element wide input and output streams, filtering of a 4-element transmission. Non-filled (white) register spaces are treated as `invalid`.

accomplishes this, and displays the steps for compressing a set of data items. The find-first-set problem is solved by "looking" at the data element fields of one bus cycle one after another, in consecutive cycles (hence additional latency). In order to not affect the throughput, the input dataword is cascaded through a triangular array (hence additional resources). Every level of the triangular array has an "inversely growing" random access buffer (meaning that the deeper into the array, the wider the buffer. If a valid element is detected, it is written to the next free position in the respective level's buffer. The buffer

itself is cascaded to the “least significant” elements of the next level’s buffer. Additionally, every level maintains a count of the number of valid elements in its buffer, such that determining the correct buffer field to write a detected valid element to is $\mathcal{O}(1)$, instead of another find-first-set (or “unset”, in this case). The count is inherited from the previous level, and incremented upon detecting a valid element.

The last stage, after the array, accumulates valid elements, and writes to the output databus. It consists of a buffer register twice the width of the databus (in theory, one field less than that), and also has a counter. Valid elements from the last array level are appended to the buffer, while the start field is determined by the current counter. The counter is updated from the last array level. If the counter exceeds the databus width, the lower buffer half is written to the output, and the upper half is shifted down. Again, all of these operations only need current counter values for parameterization, no find-first-set at any point.

The presented module introduces latency cycles in the order of the number of fields in the databus, and registers in the order of that number squared. In return, it offers high clock frequency/high throughput bus compression, and scales flawlessly with the databus width.

Engine Return Data - Buffering & Throttling

As mentioned earlier, the data back end needs to actively prevent any data loss on the return path from the engine, for instance if the memory interface reads faster than it writes. The idea here is simple. It is based on the fact that the round-trip datapath from the data back end through the engine and back has deterministic, constant latency (“there can only be so much data in the pipeline”): “Place a FIFO buffer in the return datapath. Monitor the buffer fill state. If there is less free space in the buffer than the maximum engine round-trip latency is, throttle down the memory read”. The FIFO is another XPM FIFO, this time with symmetric input and output datawidth (in contrast to the tensor operation engine module). Throttling down the memory read is done via the databus converter, which already has an externally supplied `ready` signal port to exploit, because the engine databus does not have one.

The buffer synergizes great with bus compression, provided that bus compression prepends the buffer. It is important to only write datawords with valid elements to the buffer, because otherwise the buffer would be filled up uselessly, and can stall the entire system, if not cleaned in another way. Behind the bus compressor, databus cycles (with an arbitrary number of lanes) can be filtered solely based on whether the first lane consists a valid element, written to the buffer if it does, and dropped otherwise.

5.7. Intermediary Layer - Host-Accelerator Interface

Host and accelerator communicate via two logical interfaces: instruction interface and data interface. Physically, all host-communication runs via the accelerator card’s PCIe interface. Both interfaces have in common that they require a conversion of number representation and state vector format, between accelerator and host. Therefore, after the following sections explain the kernel-side instruction and data interfaces, a separate paragraph is dedicated to discussing data conversions.

5.7.1. Instruction Interface

Logically, the instruction interface is subdivided again into a parameter interface (for operation parameters), and a control interface (for operating the accelerator). The distinction is important because these might be implemented differently. In the prototype, both interfaces eventually were implemented using the kernel register file, but an alternative implementation was tested for the parameter interface, using shared memory. The choice for register file was made due to significantly lower latency (see Section 6.4.2 for details). The idea to employ shared memory stemmed from the fact that quantum gate operations require a substantial amount of parameterization data (half of which is the base gate matrix), displayed in Table 5.2. Therefore, using shared memory reduces the number of separate data packets from about 20 register accesses (for a gate operation), to 1 memory accesses. However, the register file variant still performed drastically better than using shared (HBM) memory.

In terms of control flow, the accelerator follows the Vitis “User-Managed Kernel” flow, with implications for software and hardware. The flow essentially says that it is a kernel with a register file, which does not follow a pre-defined XRT execution protocol. An overview of the control registers is shown in Table 5.1. Next to “pure” operation control, the set of control registers also comprises any “accelerator return data” fields, like measurement result, and execution timers for profiling.

The choice for a user-managed kernel was made with latency considerations in mind. First, it allows for “non-blocking” kernel execution, where the host software only triggers a kernel operation, which is

Register	Characteristic	Function
`status_errors`		combined status bits and error flags
`status`	ro	status bits
`errors`	ro	error flags
`reset`	tr	trigger kernel hard reset
`trigger`	tr	run operation
`meas_prob`	ro, float64	p from last measurement operation
`meas_prob_sqrt_inv`	ro, float64	$\frac{1}{\sqrt{p}}$ from last measurement operation
`address_host_data`	ro, int64	address for shared on-chip data memory
`perf_count fetch_op_params`	ro, int64	raw cycle counter for profiling
`perf_count core_busy`	ro, int64	raw cycle counter for profiling
`perf_count core_finalize`	ro, int64	raw cycle counter for profiling

Table 5.1: parameter control registers. `ro`: *read-only*. `tr`: *trigger* (not backed by memory, only trigger lines that are activated by a write to that address). $64\,\mathrm{bit}$ registers (*int64*, *float64*) consist of 2 $32\,\mathrm{bit}$ registers.

Register	Characteristic	Function
`op_matrix[0][0].re`	float64	operation matrix (with following)
`op_matrix[0][0].im`	float64	
`op_matrix[0][1].re`	float64	
`op_matrix[0][1].im`	float64	
`op_matrix[1][0].re`	float64	
`op_matrix[1][0].im`	float64	
`op_matrix[1][1].re`	float64	
`op_matrix[1][1].im`	float64	
`state_idx[0]`	int32	state index 1
`state_idx[1]`	int32	state index 2
`target_qubit[0]`	int32	target qubit index 1
`target_qubit[1]`	int32	target qubit index 2
`control_qubit[0]`	int32	control qubit index 1
`control_qubit[1]`	int32	control qubit index 2
`num_qubits`	int32	number of qubits
`meas_destructive`		measurement parameter
`op type`		operation type

Table 5.2: parameter kernel registers. $64\,\mathrm{bit}$ (*float64*) registers consist of 2 $32\,\mathrm{bit}$ registers

executed autonomously, while the host CPU can prepare the following operation, for instance. This makes sense especially with quantum operations, because applying a gate has no immediate “return value”, apart from the post-operation state, which is purely on-chip - as is the case for tensor operations as well. Only measurement operations typically have to be blocking, because the measurement outcome is reported to the host. The second latency-related advantage is that a custom design can slightly reduce the number of register accesses necessary to trigger an operation. Normally, the host software would have to first check the kernel’s idle status, then potentially check an error register (and abort if any errors occurred during the previous operation), and just then trigger the kernel. A custom design can combine the status and errors register, or could even provide a combined status/errors register with a read triggering the next operation, which would reduce 2 register reads and 1 write to only 1 read. The prototype is implemented with a combined (non-triggering) status/errors register, which however is not fully functional in hardware at the time of delivery. All results are obtained with the “fallback” solution of separate status, errors, and trigger registers.

Looking at the parameter registers in Table 5.2, one might notice that only the base gate operation matrix is passed, while the accelerator’s *engine* module differentiates between several gate operation types. The kernel has a simple integrated mechanic to detect optimized gate operation types from the gate operation matrix (see Algorithm 9 below), and falls back to dense gates if nothing else applies.

A future iteration of the accelerator would likely add some extra registers, to allow for the host to immediately specify standard quantum gates. The decision for exclusively communicating via the base gate operation matrix is based on using the NetSquid simulator as the prototype front end. In NetSquid's lower layers, controlled gates are not guaranteed to retain the information of the "original" underlying standard quantum gate, while the (full) operation matrix is always available. Therefore, relying on the matrix was the option that would always be applicable, although software has to "extract" the base gate operation matrix.

Algorithm 9 Determine the *core layer* engine operation type from the single-qubit/2×2 operation matrix M, and the target qubits q_1 and optional q_2. Only the SWAP gate has 2 target qubits. except from that, the state machine tries to detect a sparsegate type, and falls back to densegate.

```
1:  procedure GetEngineOpType(M,q1,q2)
2:      if q2 then
3:          return SPARSEGATE-SWAP
4:      end if
5:      if M[0][1] == 0 and M[1][0] == 0 then
6:          return SPARSEGATE-DIAGONAL
7:      end if
8:      if M[0][0] == 0 and M[1][1] == 0 then
9:          return SPARSEGATE-REVERSE-DIAGONAL
10:     end if
11:     return DENSEGATE
12: end procedure
```

5.7.2. Data Interface

The data interface is necessary for setting and reading out state vectors. As was already shown in Figure 5.15, the prototype uses half of the on-chip HBM memory for host data exchange. The two aspects to be discussed here are the sizing, and the physical memory location.

In terms of host data exchange buffer, the prototype's data interface is implemented such that it requires a buffer of the same size as the quantum accelerator's internal quantum state memory. That is because state vectors are transferred via the buffer in full, instead of in batches, which simplifies both the involved hardware modules and software routines. In either data direction (host to accelerator, or accelerator to host), a state in shared memory takes up the same amount of memory as a state in quantum state memory. Therefore, the buffer needs to be the same size as the quantum state memory.

Considering the physical location of the shared memory, the option would have existed to use host system memory, instead of on-board memory. Several Alveo cards support accessing host memory via PCIe, among which the U55C [6]. There are two reasons for using on-chip memory (and thereby sacrificing quantum state memory) in the prototype, while a later iteration is expected to switch to host memory. First, it makes the accelerator more independent of the host system. If in any situation the system has limited main memory, for instance in a development/testing setup, the availability of the $8\,\mathrm{GB}$ on-chip buffer is not affected, and might allow exchanging larger states than the host's main memory. The second reason is FPGA portability. The host memory access feature is only available on newer Alveo cards, but older cards could still be in service for several years. The now-deprecated U280, for example, which can be seen as the U55C's direct predecessor in terms of HBM-equipped Alveo cards, is not able to access host memory. The on-chip buffer makes it easier to port the design to this or a similar card.

The host data interface is inferred the exact same way as the kernel's exclusive HBM interface, via the Vitis compiler toolchain, and therefore operated by the same AXI masters that can also be found in the quantum state memory module.

5.7.3. Data Conversion

As mentioned at the beginning of this section, instruction and data interface both require data conversion between host and accelerator. Both interfaces need a means to convert between real number representations. The data interface additionally has to deal with different state vector representation formats.

State Vector Representation

While the accelerator uses labeled-random state vectors, the software front end uses “conventional” vector representation (complex-valued arrays). The reason is generality, and compatibility with a simulator like NetSquid.

As a design decision, most of the conversion workload is handled by software, because data exchange was considered “not time-critical”. Actual states are usually only exchanged when initializing a state to anything else than $|0...\rangle$, or when observing states post-simulation. None of these operations are frequent, during a simulation. Reading out post-simulation states evidently happens only once, at maximum. “Non-standard” state initializations could theoretically appear more often, but in many simulations, a generated state is assumed to be $|0...\rangle$, and any further state preparation, or noise simulation, is modeled by gate operations.

In practice, when transmitting a vector from accelerator to host, the accelerator copies the vector to shared host memory as is, in labeled-random representation. Software then sequentially reads the vector from shared memory, and re-assembles it to conventional representation in a separate buffer, using random write accesses. When setting states on the accelerator, software writes the state to shared memory in conventional representation. Since the accelerator afterwards still needs to transfer the state to quantum state memory, and given that the state is in correct order, it causes very limited overhead to count elements, and add the labels for labeled representation as part of the process.

Real Number Representation

A second problem with data transfers is real number representation. Conventional host systems use floating-point real numbers (IEEE $64\,\mathrm{bit}$ in the prototype, and in NetSquid), but the accelerator uses fixed-point real numbers. This applies not only to the data interface for state vector exchanges, but also to the control interface, when passing the base operation matrix. Passing an operation matrix, in contrast to exchanging quantum states, is a highly frequent operation during any simulation. Accordingly, unlike state vector exchange, real number conversion was considered potentially time-critical. Therefore, the matrix conversion had to be entirely done in hardware. For consistency, the number conversion for state vector data is also done in hardware - although that has the drawback of necessitating the fixed-point to floating-point direction, which comprises a find-first-set function, and is therefore less hardware-friendly than floating-point to fixed-point.

The number converters are custom implementations. In the first iteration, they are implemented fully combinatorial, which, as expected, resulted in a critical timing path for fixed-point to floating-point conversion, by the time of delivering this thesis. However, it is projected that a pipelining register can quickly solve that problem. The fact that the conversion datapaths are fully “serial” between the converters’ input and output registers helps with pipelining, and should facilitate efficient use of the synthesis tool’s retiming functionality. Additionally, one has the backup solution of moving the fixed-point to floating-point conversion into software, because only the other direction is time-critical. Thus, although the number format conversion is implemented rather “carelessly”, it is not expected to pose a timing issue in the long run.

5.8. Software Layer

5.8.1. Architecture

As shown in the accelerator high-level design (Figure 4.1), the software layer consists of a low-level driver portion, and a user application, on which the following sections give further insights. The user application can either be a simulator on its own, or bridge to an existing simulator. Its architecture is therefore dictated by the front-end simulator. The accelerator drivers, in contrast, are application-independent, and expose the core layer interface (Section 4.2.1) to the user application.

5.8.2. Low-Level Software - FPGA Kernel Drivers

The kernel is exclusively accessed via one class, called `Nfe` (“netsquid fpga engine”), which takes care of managing physical card instances, all control and data traffic, and state vector format conversions (see Section 5.7.3). All accelerator interaction is based on XRT APIs, including initialization (which happens in the `Nfe` constructor).

An important functionality of the `Nfe` class is that it automates non-blocking accelerator execution. Section 5.7.1 explained why it makes sense for `operate` and `tensor` operations to be non-blocking,

method	checking	blocking
`measure`	yes	yes
`operate`	yes	no
`tensor`	yes	no
`clear`	yes	yes
`create`	yes	no
`readout`	yes	yes
`set`	yes	yes

Table 5.3: overview of the (standard operation) API of the `Nfe` hardware driver class. *non-blocking* here means that the function triggers the hardware kernel, and then immediately returns, while the kernel is most likely still busy. *checking* means that the function checks for the kernel to be idle, before triggering.

from a simulator front end perspective, and how that is implemented on the register file side. The `Nfe` class provides "non-blocking, but execution-safe" `operate` and `tensor` methods to the user application - which means that upon calling, the methods do block until the accelerator is idle (since there could have been an error in an earlier instruction), but then return immediately after triggering the accelerator. The `create` operation has the same behavior. An overview of the all core/low-level software API methods, and their behavior to the user application, is given in Table 5.3.

With the aforementioned functionalities, any user application software can effectively treat the accelerator as if it was a pure software back end. In addition, the `Nfe` class can be compiled in a "testing" mode for profiling purposes, which allows potentially unsafe accelerator operation (like triggering or altering parameters, while the accelerator is not idle).

A yet to be implemented functionality is the conversion from logical state indices to physical entry indices in the accelerator's memory allocation table. As explained in Section 4.4.3, either hardware or software can be responsible for this task, but the prototype design leaves it to software. Mirroring the in-hardware memory allocation table is feasible, because all core/low-level software methods are fully deterministic, in how they alter entries in the memory allocation table. Therefore, the `Nfe` class needs to contain a matching table for logical to physical indices, and every accelerator operation needs to update that table, if necessary. Additionally, it might be helpful for the `create` operation to return the logical index of the newly created state (regardless of whether state index conversion happens in software or in hardware).

5.8.3. User Application - NetSquid QRepr Layer

As indicated in Figure 5.1, at the time of delivering this thesis, there is no user application layer implemented other than a *testing front end*. However, the future integration with NetSquid was part of the design considerations. NetSquid has a modular quantum state representation back end, with a class template called `QRepr` (written in cython). It is that mechanism that also allows it to switch between operating on state vectors or on density matrices (among others). Table 5.4 shows how the `QRepr` methods can be implemented using the `Nfe` class. These routines need to be part of a yet to be implemented `NfeQRepr` class, which bridges between `Nfe` and `QRepr`, is compiled as a DLL, and included into NetSquid via cython. Next to implementing the `QRepr` interface, bridging involves `Nfe` instance handling, and quantum state index managing.

Instance handling for `Nfe` refers to the fact that there can only be one `Nfe` instance, but there is one `QRepr` instance, and hence one `NfeQRepr` instance, per quantum state. The `NfeQRepr` class needs to resolve that situation, for instance by holding an `Nfe` object as a (static) class variable, which all instances can access.

In terms of quantum state index managing, there is only little work left to do for `NfeQRepr`. Converting between qubits (`Qubit` in NetSquid) and state (`QState`) objects is done by NetSquid itself, at a higher level. Every `QState`, in turn, is linked to a `QRepr` object via class member relationship. The `NfeQRepr` class therefore only needs to make sure to call `Nfe`'s `create` in the constructor, retain the constant logical id that it got from `create` in the instance, and use it during calls to the `Nfe` instance.

`QRepr` method	(main) domain	kernel operation	comment
`__init__`	HW	`create`	
`is_valid`	SW	`readout`	kernel readout, compute in SW
`create_in_basis`	HW	`create`	(SW rotate to computational basis)
`measure`	HW	`measure`	
`gmeasure`	HW	`measure`	measure group → consecutive `measure`
`operate`	HW	`operate`	SW extracts single-qubit operation matrix, then triggers HW
`multi_operate`	HW	`operate`	consecutive single `operate`
`fidelity`	SW	`readout`	kernel readout, compute in SW
`reduced_dm`	SW	`readout`	kernel readout, compute in SW
`smart_tensor`	HW	`tensor`	SW order the states by size, and update the SW state table mirror
`tensor`	HW	`tensor`	like `tensor`, without state ordering
`discard`	HW	`measure`	re-purpose destructive measurement
`measure_discard`	HW	`measure`	destructive measurement
`__eq__`	SW	`readout`	
`__repr__`	SW	`readout`	
`__str__`	SW	`readout`	
`__deepcopy__`	SW	`readout`, `create`	SW state readout, create and set a copy of the state in HW, update state table mirror
`__copy__`	SW	`readout`, `create`	perform `__deepcopy__`

Table 5.4: realization of the NetSquid `QRepr` class API using the FPGA kernel

5.9. Prototype Overview

At the time of delivering this thesis, the prototype accelerator implementation is parameterized as follows:

- **quantum state memory**
  - **physical memory**: $8\,\mathrm{GB}$ HBM (one of two stacks), no local on-chip memory
  - **physical memory interface**: 2 512-bit datawidth AXI4 buses in dual-simplex (one read-only, one write-only)
    - resulting **engine databus width**: 4 lanes
  - **maximum number of states**: 29 (arbitrary number, related to outdated design considerations, where it originated from that a 29-qubit state uses up exactly the available $8\,\mathrm{GB}$)
  - **memory allocation algorithm**: first-fit, computation only on request
  - **engine return data buffer depth**: 2048 datawords/8192 elements BRAM ($83.5\,\mathrm{kB}$, ignoring supported BRAM data interface widths)
  - **real number format**: $27\,\mathrm{bit}$ fixed-point (1 sign bit, 1 integer bit, 25 fractional bits)
- **engine**
  - **dense gate module array depth**: 4 (quadratical array)

- **tensor operation module state buffer**: 64 elements (theoretically about $2.6\,\mathrm{kB}$ BRAM)

Functionality has been validated for monomial gates on states of up to 23 qubits, measurements and tensor operations on states of up to 22 qubits, and for non-monomial gates on states of up to 15 qubits (see Section 6.2.2).

5.10. Conclusion

This chapter demonstrated how the high-level accelerator design for labeled-random state vector computation, from Chapter 4, was realized as a prototype system on an AMD/Xilinx Alveo U55C FPGA accelerator card.

The first section identified the Alveo U55C as a suitable platform for the intended implementation. It has good availability, features on-board HBM memory, is geared towards DSP-heavy applications (in terms of DSP-to-LUT ratio), and enables straightforward host-accelerator integration via Xilinx's XRT kernel models and library. Furthermore, the section defined $450\,\mathrm{MHz}$ as the eventual system-wide target clock frequency, to maximize the HBM throughput. Hardware design guidelines were derived based on that figure. Lastly, the NetSquid simulator was chosen as the demonstration system's eventual front-end simulator, because it is reliable, allows for accelerator integration via its modular back end, and is suitable for demonstrating the full spectrum of the design's accelerator functionalities.

The following sections (Sections 5.3 to 5.4) provided the first portion of the technical discussion on the FPGA implementation. They treated aspects and components with design-wide impact (in contrast to individual modules), these being the FPGA-wide real number representation, recurring arithmetic structures, and backbone data bus design. For complex-valued state vector amplitudes, a representation in cartesian coordinates, with fixed-point components parameterized according to amplitude range and DSP port widths, was found to best exploit the available hardware. Based on that, an elementary DSP complex number multiplier was defined, using 2 adjacent, cascaded DSPs. Additionally, a lightweight dual-simplex data bus was defined for data exchange between engine and quantum state memory, that consolidates complexity in the spatially constricted memory interface, while allowing low-overhead data distribution and execution for the spread-out engine.

Afterwards, Sections 5.5 to 5.8 treated the individual modules of the accelerator hardware design bottom-up, analogous to the high-level discussion in the previous chapter. Section 5.5 demonstrated how the multiplexed engine CUs from the previous chapter translate to efficient DSP based hardware modules, that also save fabric logic by leveraging various of the DSPs' internal cascading and feedback lines. Special attention was given to the Goldschmidt-Algorithm for inverse square-root computation, whose main challenge lied in finding a hardware-friendly initial value scheme.

Section 5.6 first discussed the prototype's physical 2-channel HBM connectivity for quantum states, before elaborating on quantum state data alignment in memory. Afterwards, the prototype's find-first-fit memory allocation state machine was introduced. Additionally, the future importance of state distribution in multi-channel memory was exemplified, although not considered in the prototype. In terms of data movement, the section explained the necessity of bus conversion, and data buffering and filtering, between quantum state memory and engine. It was introduced how the data back end achieves the required data filtering in a scalable way, without any throughput loss, and how a BRAM FIFO provides buffering and data loss prevention for round-trip data.

Section 5.7 first explained the accelerator's register file-based host instruction interface. It was then highlighted how the decision to employ a large on-chip buffer for host data exchange, in the prototype, does cost significant quantum state memory, but greatly increases the flexibility to port either the whole accelerator to a different host system, or the implementation to a different Alveo FPGA. Additionally, the section highlighted data format conversion between accelerator and host, where time-critical real number conversion was implemented in hardware for higher performance, while software is responsible for non time-critical state vector format conversion for simplicity.

Section 5.8 described how the software layer consists of one front end-independent class for accelerator abstraction, and simulator-dependent higher layers. It is explained how NetSquid can be integrated via its `QRepr` back end, going through the full API, by means of a "bridging" class.

Lastly, Section 5.9 gave an overview of the hardware system's parameterization, at the time of delivering this thesis.

6 Results

The previous chapter illustrated how the high-level accelerator design was implemented on an Alveo U55C Data Center FPGA card, and equipped with a low-level software interface for host-accelerator interaction. This chapter discusses any findings that result from synthesizing, validating, and benchmarking that implementation, in the parameterization given in Section 5.9.

Naturally, the most important aspect to study is the computation performance of the implemented quantum state vector operations. In this case, performance is interpreted as both accuracy, and execution time. However, prior to presenting these results, Section 6.1 gives the current FPGA implementation figures (resource usage, clock frequency). The reason for that order is that the implementation metrics are essential information to put the aforementioned results into perspective, especially for execution time. Following these metrics, Section 6.2 is dedicated to validation and accuracy, after which Section 6.3 discusses execution time results. The results incorporate both an in-detail comparison between the accelerator and a software baseline, as well as a profiling of the accelerator execution in particular. The elaboration concludes with future speedup projections for both the accelerator in isolation, and as a NetSquid back end. Lastly, Section 6.4 discusses any auxiliary findings that fit best into a separate discussion, although the involved modules are part of the quantum state vector operations. Namely, auxiliary results are given on parameterization and accuracy of the isolated FPGA Goldschmidt-Algorithm implementation (Section 6.4.1), and in the form of a latency study on accelerator operation parameterization methods (Section 6.4.2).

All FPGA experiments were conducted using AMD/Xilinx Alveo U55C cards in AMD's Zurich "Heterogeneous Accelerated Compute Cluster (HACC)" [40].

6.1. Implementation

As mentioned in the previous paragraph, this section evaluates the prototype's FPGA implementation figures, in order to establish the necessary context for discussing application performance results in Section 6.3.

Clock Frequency and Critical Paths

At the time of delivery, the kernel is operating at a maximum clock frequency of $224\,\mathrm{MHz}$. The clock is supplied via the Vitis compiler toolchain, which after Place & Route (PnR) automatically scales the clock frequency down as much as it needs to avoid timing violations.

In the prototype implementation, the dominant critical paths are the combinatorial fixed-point to floating-point converter - which was expected - and the engine dense-gate label filtering. The former can likely be resolved by means of a pipeline register, and potentially retiming, as indicated in Section 5.7.3. About the latter, it must be strengthened that the dense-gate module implementation uses the legacy algebraic element update formulas (Equations (C.5) to (C.6)). It is considered likely that the simpler formulas (Equation (3.9), Equation (4.1)), that utilize pairwise gate operation, can significantly improve this path's performance. Otherwise, it is completely possible to introduce an additional pipelining cycle in the filtering circuit, to remove remaining issues.

While there is no clear figure for the clock frequency improvement with the aforementioned measures, it is a promising fact that only two, extremely local paths account for most of the currently failing endpoints,

while both paths have a straightforward potential solution. However, future code development might uncover further need for intervention when nearing the intended $450\,\mathrm{MHz}$.

Resource Usage

In terms of FPGA resource usage, the accelerator, in its current parameterization, exhibits the expected, relatively small footprint. In fact, at the time of delivery, the Vitis hardware shell consumes roughly 6-8 times as many on-chip resources as the accelerator kernel, with the exception of DSPs. Table 6.1 and Table 6.2 give an overview of the resource usage for the whole design, and the accelerator in isolation (SLRs 1 and 2 do not contain any kernel resources so far).

	SLR0			
	Units		Usage (%)	
	all	kernel	all	kernel
CLB	15695	5399	28.56	9.82
CLB LUTs	71625	27410	16.29	6.23
CLB Registers	89427	17391	10.17	1.92
BRAM Tile	84	24	12.5	3.57
UltraRAM	0	0	0	0
DSP	117	117	4.06	4.06

Table 6.1: FPGA resource usage for SLR0 (in absolute units, and in percentage of the available resources per SLR). *kernel* are the resources for the vitis kernel only (labeled *HW* in Figure 4.1). *all* are the cumulated resources for the *kernel*, and the vitis compiler-generated shell. resources with low relevance for kernel development are omitted (Clock Management, Gigabit Transceivers)

	SLR1		SLR2	
	Units	Usage (%)	Units	Usage (%)
CLB	9189	17.02	6280	11.63
CLB LUTs	58135	13.46	25694	5.95
CLB Registers	66295	7.67	35528	4.11
BRAM Tile	90	13.39	50	7.44
UltraRAM	0	0	0	0
DSP	0	0	4	0.13

Table 6.2: FPGA resource usage for SLR1 and SLR2 (in absolute units, and in percentage of the available resources per SLR). All resources belong to the vitis compiler-generated shell, the *kernel* (see Table 6.1) only occupies resources in SLR0.

In terms of kernel resource usage, it is insightful that, although the design has a considerable level of pipelining, the relative and absolute LUT usage is higher than the register usage. This observation is an indication that, should additional registers be needed for clock frequency optimization in any computing module, these registers might even be available ”for free“, because the respective CLBs are already in use to provide the necessary LUTs.

Looking at the non-kernel resource usage, one notices that the Vitis-inferred logic, on its own, occupies about $10\,\%$ of the available LUTs, $15\,\%$ of the registers, and $10\,\%$ of the BRAM tiles. While not dramatically high, these are substantial figures to be aware of, when scaling up the accelerator. Even more so since the Vitis shell provides the HBM memory interfaces, and therefore grows in size with more HBM channels being used. On the positive side, the generated hardware does not consume any DSPs, which keeps these fully available to the accelerator.

6.2. Validation

Before doing performance measurements, correct functionality of the implementation must be ensured, with respect to the implemented mathematical operations. This section presents the prototype evaluation results in that domain. There are multiple reasons to spend a separate section on this portion of testing. First, it gives a clear overview on which scenarios have been tested, and which have not. Second, and

more importantly, there are multiple mathematically “noisy” components in the system (not referring to electric noise) - namely the narrow fixed-point number representation on its own (in combination with the truncated second DSP multiplier operand), the inverse square-root computation via Goldschmidt-Algorithm, and determining the measurement result via LFSR.

6.2.1. Test Data

For validation purposes, sets of “static” test vectors were generated for several test scenarios, using NetSquid (in software, with its existing state vector `QRepr` back end). “Static” means that the vectors are not re-generated between iterations of the same test. This first helps with flexibility during development phases, because this way, NetSquid does not have to be installed on the same system that hosts the accelerator. Additionally, not all validation scenarios benefitted from “live” generated (random) test data, as the following paragraph explains.

Test vector sets were set up both per operation/gate type, and for mixed consecutive operation types, as much as feasible. The per-operation tests were manually set up, for better traceability during hardware debugging. As an illustration, numbers with only one non-zero bit (in binary representation) are dramatically easier to track in logic simulation with multiple, partially custom number representations, than random numbers. Tests with consecutive operations use manual, initial states, and a randomized (constrained) operation queue. In summary, the following tests were set up:

- **Debugging**

  manual initial vectors and operations

  - single non-probabilistic engine operations (diagonal/antidiagonal/swap/dense gates, tensor operations)
  - measurement operations - observing reported measurement probability, inverse square-root result, and post-measurement state

- **Validation**

  - multiple (up to 100) consecutive non-probabilistic engine operations

    manual, scrambled initial vectors, randomized operation queue (randomizing both operation type, and parameters like target states, target qubits, control qubits)

    post-operation state validation after every operation
  - measurement operations - observing average measurement result across a large number of same-measurement iterations

    excluded from consecutive non-probabilistic operation testing, because the probabilistic nature would complicate post-operation state checking enormously.

6.2.2. Functionality

The system proved to operate correctly in hardware up until a state size of 22 qubits (23 qubits for monomial gates). Operations larger than that, at the time of delivery, cause the accelerator to stall in hardware for a yet to be diagnosed reason. The problem does not occur in Vitis hardware emulation.

Additionally, correct operation of the engine’s dense gate module was only verified for states of up to 15 qubits, for execution time reasons. The respective elaboration can be found in Section 6.3.3.

6.2.3. Fidelity and Accuracy

This section discusses the evaluation of the mathematical “correctness”, or reliability, of the accelerator. As stated in Section 2.3.2, all fidelities in this section are square-root fidelities.

Non-probabilistic Engine Operations

The first finding is that the fixed-point number representation yields good accuracy, in line with the expectation in Section 5.3. The data that underlines this conclusion comes from two experiments - one in hardware, one in simulation - running all engine operation types except for measurements. First, in an accelerator run on initial states of up to 4 qubits, after 15 operations (among which 3 tensor operations), the worst-case post-test state fidelity was $0.999\,951$. In a similar simulation, that additionally contained fixed-point/floating-point conversion before and after every operation, the worst-case post-simulation state fidelity after 100 operations (5 tensor operations) was $0.999\,797$. Since the second figure includes

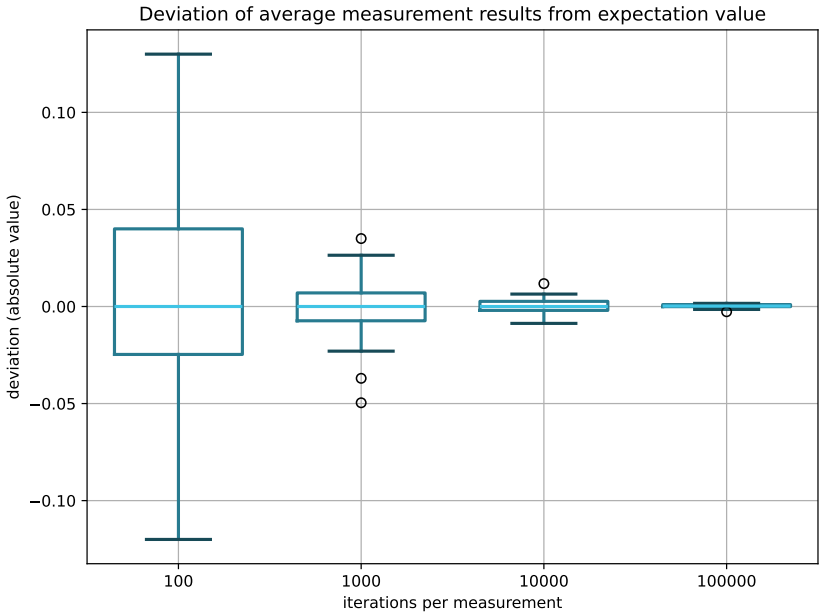


Figure 6.1: Accuracy of the empiric measurement outcome probability at different sample sizes, on a set of 7 fixed test states and measurement operations. Data acquired by doing multiple runs of performing the same operation *iterations per measurement* times consecutively, and averaging the measurement outcomes per run. *Deviation* always refers to the "correct" (analytical) probability for the specific state and measurement.

per-operation number format conversion noise, which does not appear in hardware execution, both tests indicate that the step from $64\,\mathrm{bit}$ floating-point to $27\,\mathrm{bit}$ fixed-point numbers does not cause any serious fidelity loss.

Post-Operation State Fidelity

In terms of post-operation state fidelity, one can observe a significant impact of the numeric inverse square-root computation. In experiments on states of up to 4 qubits, the average post-measurement state fidelity (compared to the NetSquid-generated reference date) after just one measurement was $0.997\,083$, with the negative outlier being $0.993\,863$ (mix between destructive and non-destructive measurements). While that fidelity is still "usable", these findings suggest that it is desirable to increase the system's accuracy for post-measurement states. It is relatively certain that the problem lies in the inverse square-root computation, since the remainder of the post-measurement state computation functions almost the same as the monomial gate module, which in the previous paragraph has been proven to be unproblematic.

Section 6.4.1, which focuses on the Goldschmidt-Algorithm implementation results, proposes approaches to enhance the achievable accuracy, among which increasing the internal multiplier width. In case further action is required, the option exists to extend the increased datawidth to the post-measurement state computation DSP array, such that both input state elements and normalization factor can be used at full available precision. Although that doubles the DSP count for post-measurement state computation, that is an acceptable hardware cost, because it is an insignificant increase compared to the dense gate array's DSP consumption.

Average Measurement Outcome

Lastly, the average measurement outcome was analyzed, and matched against the "correct", analytical value. The results, in terms of absolute value deviation, are displayed in Figure 6.1. The plot shows how, for a large enough sample size, the average measurement outcome converges towards the correct value. However, one can observe a minimal positive bias. The likely reason lies in a bias in the LFSR's $48\,\mathrm{bit}$ counter values during the early portion of its period time. The hardware was re-initialized for every experiment, meaning that also the LFSR had the same value at the start of every experiment. In theory, the impact decreases with the length of the experiment, and thus the number of measurement iterations

in the experiment. The differences however are insignificant, because the longest experiments took several minutes, while the LFSR's period time is a bit over 12 days, at the current system clock frequency. Concluding, the data shows that the LFSR-based entropy source generally offers good randomness, but that a smaller LFSR, with a shorter period time, might be favorable for bias-free operation.

6.3. Performance

This section discusses the results of any computing performance-based experiments on the prototype, after the previous section established reliable accelerator operation. First, Section 6.3.1 lays out the evaluation setup - referring to test scenarios and hardware specifications - for both the accelerator prototype, and a software baseline. Afterwards, Section 6.3.2 points out specific characteristics in the baseline execution, which will occur repeatedly in the discussion of the prototype results afterwards. Figures obtained from prototype execution are split up into overall computation performance, compared to the baseline (Section 6.3.3), and profiling of the hardware execution in isolation (Section 6.3.4). Lastly, an outlook is given on the potential performance when scaling up the accelerator to using all 32 HBM channels - first in terms of pure quantum state computation (Section 6.3.5), and then in the context of using NetSquid as a front-end user application (Section 6.3.6).

6.3.1. Setup

In terms of testing setup, this section first introduces the hardware and software frameworks used for execution time evaluation. Additionally, the general guideline for measurement operation patterns is established. This will be done in the following paragraphs. The paragraphs below, on *Baseline* and *Accelerator*, will each, for their respective component, explain how execution time exactly is measured, what is used as test data, and how the test instructions are set up.

For execution time measurements, it is imperative to establish a suitable baseline. Otherwise, the accelerator results can not be put into any meaningful perspective. Analog to functionality validation, NetSquid's quantum state vector back end served as the baseline simulator. As the computing platform for the experiments, the NetSquid development group kindly provided access to their HPC multi-core CPU node. It is equipped with an Intel Xeon Gold 6230 CPU, and $120\,\mathrm{GB}$ of RAM. All performance baseline measurements were performed on that machine, in idle state (no other noteworthy processes active). Nonetheless, it needs to be pointed out that most baseline performance measurements reflect single-core performance. However, no such restrictions were enforced, the CPU was free to use any number of cores. The information does not stem from code inspection, but solely from visually monitoring core activity during execution.

Considering worthwhile data to collect, the intuitive answer is performing sweeps of the same operation (type), applied to different state sizes. In addition to that general idea, it was decided to separately observe "different flavors" of operations, where applicable. For quantum gate operations, that means distinguishing between operations with 0, 1, or 2 control qubits. Measurements were divided between destructive and non-destructive measurements. For tensor operations, it was studied whether or not the order matters, in which states of different size are passed to the operation ("does it have an impact which state is the smaller one?").

Baseline

Execution time is measured in python, at NetSquid's `qubitapi` level. Therefore, any upper-layer semantics of the simulator are omitted, like simulating Quantum Networking nodes and protocols, or quantum algorithms. There is however a bias to using `qubitapi`, because that makes resolving `Qubit` objects into `QState` objects, and then `QRepr` objects, part of the timed execution path. The decision to nonetheless use `qubitapi` originated from the fact that it is the lowest level in NetSquid that is written in python, instead of cython. Measuring at any lower layer would have required altering NetSquid code itself, and re-compiling (NetSquid is distributed pre-compiled), while `qubitapi` meant just using an existing API. Additionally, although that was not specifically profiled, the introduced overhead is not expected to have a considerable impact on the measurements, because all additional steps consist of simply resolving class member relationships, in mostly compiled code.

The test data consisted of a set of vectors with partially random amplitudes, with only non-zero elements, to prevent optimization effects.

The instruction queue consists of one instruction for each "flavor", per state size, that are applied

one-after-another. Instructions are determined at random (with respect to the respective operation type), as are target and control qubits. Instructions are randomized for every test iteration.

Accelerator

Execution is measured at multiple levels, to allow precise profiling. Firstly, the total accelerator execution time is measured, from software's perspective. This is achieved in `Nfe`, with timestamps exactly around the XRT API calls that control the kernel. In NetSquid terms, this would roughly be `QRepr` level. Still, if implemented as proposed in Section 5.8.2, all steps between `qubitapi` and `QRepr` can be achieved by resolving compiled class member relationships, and therefore hardly contribute to the execution time. For all measurements, the `Nfe` class is compiled in "testing" mode, which makes all kernel function calls blocking (see Section 5.8.2).

In addition to the total execution time, multiple splits are measured, to get a better understanding of the kernel execution, and the software-hardware communication. The first portion is the *setup/trigger* time, which is the time that software spends on passing operation and parameterization to the kernel (via register file), and sending the trigger signal (also register file). The second portion is the hardware-side *kernel* execution time, from trigger to becoming idle again, measured via a counter in the accelerator.

The test data consisted of a set of $|0...\rangle$-initialized states. The accelerator engine treats every amplitude equally, so the state vector content is irrelevant to the computation time, which eliminates any need for data randomization or scrambling.

Like for the software baseline, the instruction queue consists of one randomized instruction for each "flavor", per state size, that are applied one-after-another. However, in contrast to software, instructions are not randomized per iteration, the instruction queue is fixed. That is for the same reason as the simplistic state vectors, there is no caching around the accelerator, and any same-type operation executes exactly the same. Regardless of the exact operation, differences between iterations only originate from variations in host CPU execution, PCIe communication latency, and on-chip HBM access latency.

6.3.2. Baseline Execution Characteristics

Before looking at the performance data of the accelerator experiments, it is helpful to get a good understanding of the baseline execution in software. In colloquial language: "What is the accelerator prototype competing against?". To give the conclusion first, the software execution times show the behavior of sparse matrix multiplication, significantly aided by cache memory for state sizes of up to 20 qubits. This is a realistic assumption, for the following reasons. On the one hand, NetSquid itself has the ability to apply either sparse matrix multiplication or generic matrix multiplication, next to numpy and cython optimizations possibly also playing a role. On the other hand, the Xeon Gold 6230 CPU has a substantial $27.5\,\mathrm{MB}$ of total cache memory, which was entirely available to the simulation, and therefore can play a role up to relatively large data structures. The following paragraphs discuss the observations that led to the formulated conclusion, starting with sparse matrix multiplication, and continuing with cache memory effects.

The most apparent characteristic that underlines the assumption of sparse matrix multiplication is that the baseline execution time for gate operations eventually scales (almost) linearly in the state size (number of elements), but clearly not quadratically. This practically eliminates the possibility of generic matrix-vector multiplication. Additionally, the data shows that the computation time is 3-4 times slower for dense gates, than for monomial gates. Part of the reason, in sparse matrix multiplication, is the number of required multiplications (or, non-zero elements in the full operation matrix), which doubles. Since that explains only about half of the difference, it is expected that the procedure of assembling the operation matrix, which becomes more complex as well, accounts for the other half. Lastly, the previous section mentioned that the CPU was only using one core during matrix multiplication. While compilation-dependent, that behavior matches sparse matrix multiplication better than generic matrix multiplication. Both are parallelizable, from a pure calculation standpoint, but generic multiplication benefits more, due to predictable dimensions and fully sequential memory accesses. Sparse multiplication inherently means random memory accessing, which is detrimental for a multi-node/multi-core execution. Therefore, single-core matrix-vector multiplication in a reasonably optimized codebase (numpy, cython) can be seen as a weak indicator for a "sparse-like" multiplication.

In terms of cache memory effects, all baseline execution data shows a significant jump in execution time from 20-qubit to 21-qubit states. This data is displayed in Figures 6.2 and 6.4 to 6.6, and is discussed

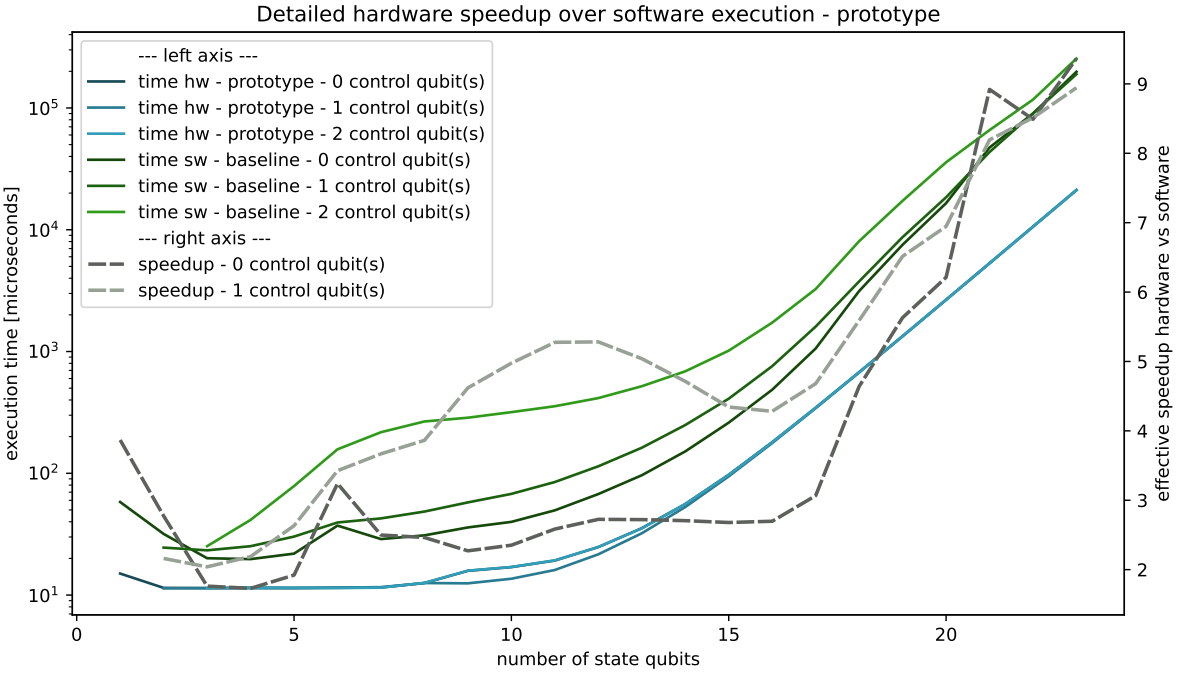


Figure 6.2: Speedup hardware over software for *monomial* gates

in more detail in the following section. The jump is sustained for state sizes from 21 qubits onwards, meaning that it is not an outlier. Rather, it is a one-time kink in the graphs, in an area where these otherwise show relatively constant scaling. The likely explanation is that exactly during the transition from 20 to 21 qubits, the CPU runs out of cache memory to hold an entire state vector - which, combined with random accessing, would lead to a significant increase in non-predictable DRAM accesses, and thus a considerable performance drop. This hypothesis is supported by the involved cache and vector sizes: A 20-qubit state vector takes up $16\,\mathrm{MB}$, a 21-qubit state requires $32\,\mathrm{MB}$, with $27.5\,\mathrm{MB}$ of total cache memory. The clear visibility of this effect is thanks to the fact that the benchmark process ran on an otherwise idle CPU, such that the actually available hardware resources were a constant across all experiments.

6.3.3. Results - Accelerator Performance

The first category of results to be presented is the raw total operation execution time, fully measured from software, as was explained in the previous sections on *Baseline* and *Accelerator*. Figures 6.2 to 6.6 below displays the measured data for monomial gates (up to 23 qubits), measurements and tensor operations (both up to 22 qubits), and dense gates (up to 18 qubits). The maximum state size for dense gates arises from execution time limitations, the maximum state size for the other operations are the maximum currently functional state sizes (see Section 6.2.2). Per operation, the graph behavior is discussed in the following, with respect to noteworthy ranges of state vector sizes. Recurring effects are only analyzed once.

The analysis will loosely use the terms *large* and *small* states, when pointing different effects in the execution time graphs. In the remainder of this chapter, if not stated otherwise, small states are states with roughly up to 8-10 qubits, and large states are states with 18-20 or more qubits.

Monomial Gates

Analogous to the implementation, the analysis starts with monomial gates (Figure 6.2). Relevant observations are collected in a first list, and analyzed afterwards:

- **General Speedup**: There is a clear speedup over software execution, at any state size. The speedup factor generally increases with the state size. It stays within a corridor between 2 and 3.5 (for non-controlled gates), up to a state size of 17 qubits, after which the speedup increases considerably, up to a factor of 6 at 20 qubits, before jumping to factor 9, for 21-qubit states and

larger. Additionally, for very small states (up to 9 qubits), the accelerator execution time exhibits a plateau, while the software execution time at least slightly increases.

- **Controlled Gates**: The number of control qubits is irrelevant for the accelerator execution times. Software seems to consistently lose performance with an increasing number of control qubits. The performance loss with 1 control qubit is roughly factor 2 up to a state size of 16 qubits, and then decreases until it becomes insignificant around 21 qubits. 2 control qubits is significantly more problematic, with a performance drop by factor 10 at states of 8-10 qubits.

The above observations can be analyzed one-by-one:

- **Plateau-Behavior for Small States**:

  Every operation on the accelerator has the constant overhead of first setting up parameters and triggering the accelerator, and afterwards querying the status register to register execution completion. Additionally, state accesses in HBM have an inital latency in the order of at least 100-150 cycles (or $0.45\,\mu s$ to $0.67\,\mu s$, at the current clock frequency). The hypothesis is that this constant overhead completely dominates the actual computation time, for small states. The analysis in Section 6.3.4 shows that this is indeed the case.

- **Higher Speedup for Large States**:

  The jump in the speedup happens just after the graphs for both hardware and software have stabilized into visually linear behavior (in the logarithmic plot). This behavior suggests that the computation time is dominated by the pure computation performance, while for smaller states, execution overhead and initial memory load times might play a bigger role. The jump itself is caused by a sustained kink in software computation time, when states go from $1\,\mathrm{MB}$ to $2\,\mathrm{MB}$ in size. It is assumed that this represents a physical threshold in the CPU cache, potentially a cache level boundary, such that the CPU loses an advantage that it has over the purely DDR-based accelerator. In other words, the systems get one step closer to comparing raw “computation and main memory” capabilities.

- **Increasing Speedup for Large States**:

  For understanding the increasing speedup, one needs to study the scaling behavior of the two computation methods. Figure 6.3 zooms in on the computation time scaling during accelerator and baseline execution, for states of 16 qubits or more. Both graphs seem to be converging towards linear scaling.

  The accelerator converges in a consistent way. That is likely because the computation itself scales linearly, and the system has no components that would introduce significant variance. Therefore, the only noteworthy difference between state sizes is the decreasing share of the time-constant setup and finalization overhead. For the software baseline, the tendency is the same, but the graph is less consistent, and it is further away from linear scaling, by a minimal amount. The convergence itself might seem surprising, given random accessing in a DRAM-based system. However, as Section 6.3.2 explained, it is likely that cache memory still plays an influential role, on these state sizes. It is uncertain how the software baseline behaves for states beyond noteworthy cache effects. Statistically, the probability for a page miss increases, due to “scattered” accesses across a larger DRAM region, which would lead to worse scaling behavior for larger quantum states. The inconsistency of the graph could be explained with pairwise gate operation effects. The smaller the target qubit index, the closer together two “pairwisely” processed input vector elements are (software uses conventional array vectors) - which increases locality in consecutive multiplications in sparse-matrix operation, leading to a higher DRAM page hit ratio, or a higher cache hit rate into higher-level caches. Since the operation queues were randomized, it is possible that in the experiments, certain state sizes were affected by large target qubit indices more than others.

  Concluding the discussion on scaling behavior, the data suggests that the software baseline’s random memory access patterns make it infeasible to fully match the scaling of the accelerator’s sequentially accessed DDR, but that CPU cache helps to mitigate the disadvantage up to relatively large states. It needs to be pointed out that, with an exponential increase in the problem size, that seemingly subtle disadvantage is enough to cause severe performance loss, or speedup gains, depending on the perspective. Provided that the hypothesis on cache effects is true, the CPU

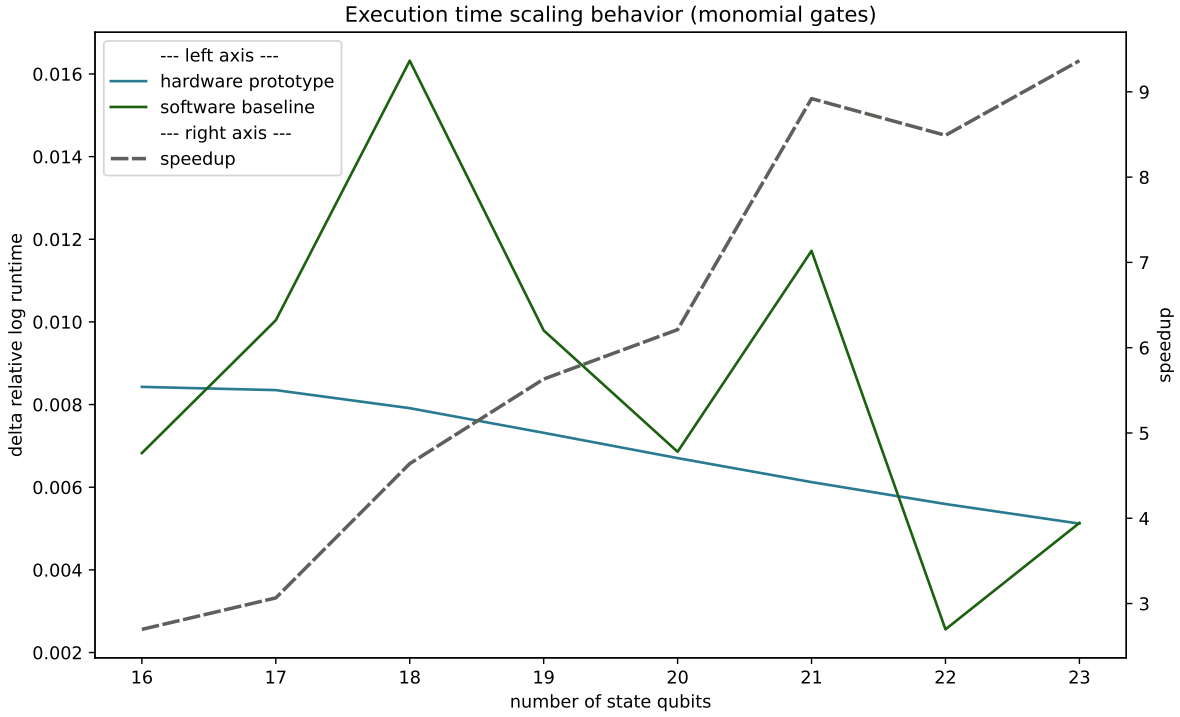


Figure 6.3: Comparison of execution time scaling between software baseline and hardware prototype, for relatively large state size. *delta relative log runtime* represents the "deviation from linear scaling, in the vector size". Calculation: Base 10 logarithm of the execution time (to eliminate exponential vector size scaling), divided by number of qubits (for normalization), minus the same value of the previous state size (for a delta). Alternative interpretation: "Difference in normalized runtimes between state sizes". A value of 0 denotes perfectly linear scaling.

would face additional problems at larger states, or as soon as more cores are active (which would both use up cache space, and cause cache coherence overhead).

- **Software Performance Loss for Controlled Gates**:

  The element update formulas eliminate the need to ever compile the "full" operation matrix, and apply controlled gates "on-the-fly", without any overhead. An implementation based on conventional matrix multiplication, which entails sparse matrix multiplication, does not have this advantage. It needs to "compute the controlled qubit index into the non-controlled operation matrix", which can be non-trivial, mathematically. The computation itself however would be no different from non-controlled gates. The resulting assumption is that the performance loss arises from greater complexity in assembling the operation matrix, prior to carrying out the operation. This can also explain why the performance loss diminishes with larger states - at least for single-controlled gates - if one assumes that the gate assembly overhead does not grow at the same rate as the computation time, and thus loses significance. While the drastically worse performance of double-controlled gate remains not fully understood, a possible interpretation is that the additional control qubit just makes it "more likely, if not almost guaranteed" to encounter a mathematically more complex operation during gate assembly. As pointed out, the accelerator is, by design, immune to any of these problems.

Measurements

The next portion of the discussion focuses on measurement operations, for which Figure 6.4 displays hardware and software execution times. It yields the following major observations:

- **General Speedup**: There is again a clear speedup over software execution, at any state size. In contrast to software, the speedup decreases from small to medium-sized (18 qubits) states. Small states are processed 10 or 25 times faster (depending on destructive or non-destructive/preserving measurement). This speedup factor drops to 4/8 (depending on the measurement type) around 18-19 qubits, and then jumps to 7/12 at 21 qubits.

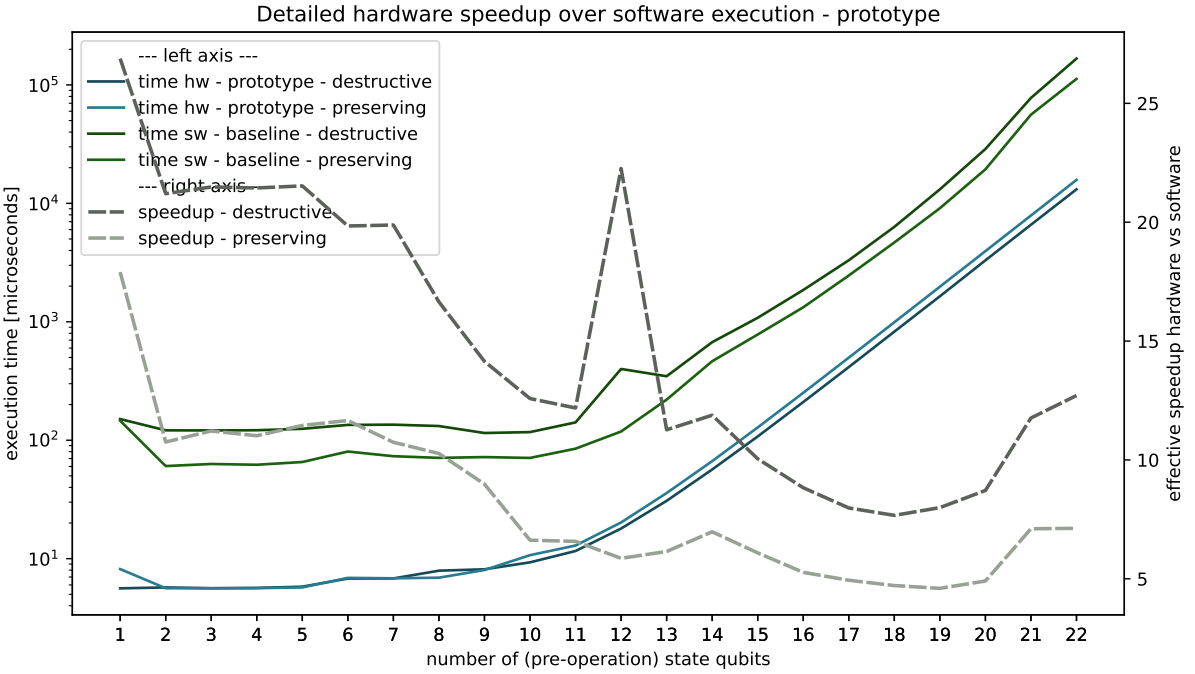


Figure 6.4: Speedup hardware over software for *measurement* operations

- **Measurement Type**: It is apparent that in software, non-destructive measurement is faster by about factor 2, across all tested state sizes. In hardware, there is no evident difference up to 11-qubit states, from where on destructive measurements execute slightly faster.

The analysis of the aforementioned observations is as follows:

- **Speedup Drop up to 19 Qubits, Increase past 19 Qubits**:

  The speedup behavior can be better understood if one compares the absolute speedup numbers to the previously discussed monomial gate operations (Figure 6.2). From 19 qubits onwards, non-destructive measurements and non-controlled monomial gates see very similar speedups in hardware. Due to the architectural similarity of both units in the accelerator engine, it is assumed that the same aspects hold that were discussed for monomial gates, under *Increasing Speedup for Large States* (in conjunction with the baseline scaling behavior, Section 6.3.2).
- **Enormous Speedup for Small States, Drop up to 19 Qubits**:

  The fact that from 19 qubits on, the behavior resembles that of monomial gates, suggests the following explanation for the enormous hardware speedup on small states: The non-scaling portions of the operation, whose contribution to the overall computation time diminishes with increasing state size, execute dramatically better in hardware. These portions comprise determining the measurement outcome, and computing the post-measurement state normalization factor. It was not determined which operation has the larger impact, but for both of them, it is understandable how they would be significantly faster in the accelerator. When determining a measurement outcome, software needs to first activate some type of RNG-implementation, draw a sample, and then compare two floating-point numbers. In hardware, with fixed-point real numbers, and the LFSR always active “in the background”, all of that is done within one clock cycle. Considering the normalization factor, that operation is not particularly unproblematic in hardware. However, the dedicated Goldschmidt-Algorithm circuit, at a prototype computation time of roughly $93.75\,\mathrm{ns}$, appears to still significantly outperform a CPU, which needs to rely on generic ALU operations.
- **Slower Destructive Measurements in Baseline**:

  The exact reason for the worse performance of destructive measurements in software, compared to non-destructive measurements, could not yet be determined. Since the disparity is constant across all state sizes, it is likely related to the NetSquid implementation, rather than to a machine characteristic.

- **Faster Large-State Destructive Measurements in Accelerator**:

  Firstly, in contrast to software, the accelerator does not experience an implementation-induced difference between destructive and non-destructive measurement, because the implementation is practically the same for both variants. Up to writing out the post-measurement state, they are identical in terms of execution time. Therefore, the slightly better performance of destructive measurement is expected to stem from slower writing than reading in the HBM memory interface. If that were to be the case, the write channel would be consistently "falling behind" the read channel, which at some point makes it necessary to regularly throttle the read channel. In destructive measurements, there is only half as much data written as is read, making that problem highly unlikely to occur. This reasoning also explains why no difference can be found for small states, while the gap stabilizes for large states. For small states, the buffer in the engine return data path can mask a slower write channel. For large states, the buffer size (2^{13} state vector elements) becomes insignificant in comparison - while the computation time is essentially proportional to the memory throughput, which is reduced by the constant disparity between read and write channel.

Tensor

For tensor operations, the hardware execution times were divided into two categories, due to significantly different performance. Early experiments showed that the accelerator's tensor operation execution time heavily depends on the relation between the "registered" and "streamed" state (see Section 4.3.2). Namely, a large registered state (with respect to the post-operation state size) led to a significantly worse performance. Therefore, Figure 6.5 contains one graph for *small/medium* registered states, and one graph for *large* registered states. These were obtained by running the following three experiments per post-operation state size, with different input state size ratios. Denoting the post-operation state size as n_{out}, the input state sizes $n_{\text{in,reg}}$, $n_{\text{in,stream}}$ were

small/medium registered state:

$$n_{\text{in,reg}} = 2, \qquad n_{\text{in,stream}} = n_{\text{out}} - 2 \tag{6.1}$$

$$n_{\text{in,reg}} = \lfloor n_{\text{out}}/2 \rfloor \qquad n_{\text{in,stream}} = \lceil n_{\text{out}}/2 \rceil \tag{6.2}$$

large registered state:

$$n_{\text{in,reg}} = n_{\text{out}} - 2, \qquad n_{\text{in,reg}} = 2 \tag{6.3}$$

$$\tag{6.4}$$

Between both experiments with a *small/medium* size registered state, there was only a negligible increase for *medium* over *small*, hence this way of grouping the three experiments into two categories.

From an application-based standpoint, it is valid to introduce this grouping, because the results between two input state orders are equivalent (although not equal). Simulation scenarios generally allow for ordering the states internally, unless a state readout is required in an exact qubit order. In that case however, as an alternative to not order states before merging, it is possible to keep track of state merging operations, and re-order the elements of the readout state accordingly in software. The important implication of this argumentation is that the less favorable category of *large* registered states can be ignored for the accelerator execution time and speedup, because every combination of input states can be ordered to have a registered state that is not larger than the streamed state.

- **General Speedup**: There is again a clear speedup across all tested state sizes, provided that the states are ordered by size. It gradually declines from factor 5 to factor 2, over a state size span of 3 to 20 qubits. From 20 to 21 qubits, it jumps to factor 5 again.
- **Large Registered States**: Scenarios with larger registered than streamed states perform drastically worse on the accelerator, from post-operation state sizes of 7 qubits on.

These observations are explained as follows:

- **General Speedup**:

  The behavior for large states again displays the benefit of sequential DDR accesses. The speedup jump from 20 to 21 qubits is due to the previously discussed kink in the baseline performance, and

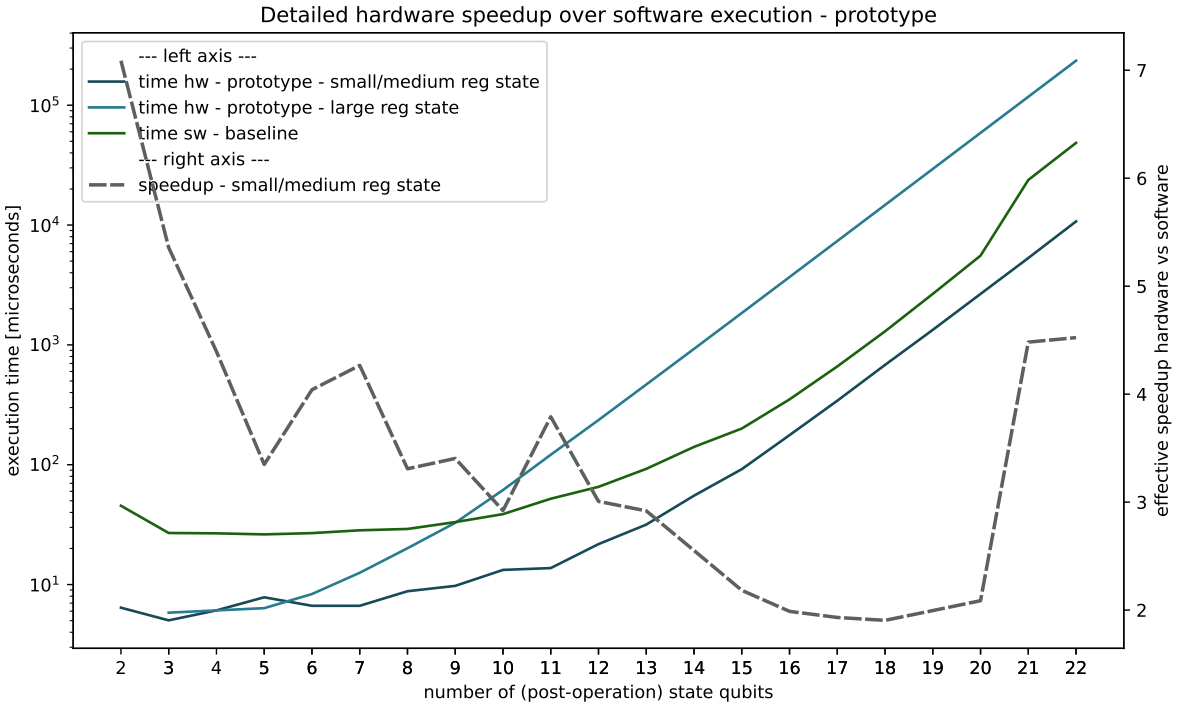


Figure 6.5: Speedups hardware over software for *tensor* operations with *small/medium* registered states, with execution times for *small/medium* and *large* registered states (see Equations (6.1) to (6.3) and respective paragraph for definitions of *small*, *medium*, *large*)

does not require any further explanation. Considering small states, the dedicated state buffer is considered to outperform cache memory in the CPU. Since the buffer does not scale, it is also the reason for the higher speedup compared to medium-sized states, similar to the static computation steps in measurement operations. Since the buffer (currently) fits 6-qubit states at maximum, and due to Equation (6.2), all post-operation state sizes of up to 12 qubits only require one memory read for the registered state. Beyond that figure, the exponential scaling in state vector size quickly dominates, which diminishes the buffer size advantage over time.

- **Large Registered States**:

  A larger registered state means that, in the proposed computation scheme, the memory back end needs to perform more separate transactions, with also a higher number of alternations between two states, thus unrelated memory locations - accompanied by a less significant increase in the total number of state elements to read. As an example, the situation looks as follows for joining a 2-qubit and a 10-qubit state: If the 2-qubit state is registered, there is 1 read to a 4-element state first, to fill the buffers. Afterwards, there are 4 reads to a 10-qubit/1024-element state, for streaming. That accumulates to 4100 total elements read, 5 total transactions, and 1 alternation between states in memory. In the reverse case, reading the 10-qubit state needs to be done in batches of 64 elements (the currently implemented tensor module buffer size), followed by 64 reads of the 4-qubit state, for every batch. By the same argumentation as above, that leads to $1024 + 1024 \cdot 4 = 5120$ elements read, $1024/64 + 1024 = 1040$ total transactions, and 31 alternations between states in memory. It is evident that this access pattern leads to a drastic decrease in average DDR throughput, compared to the first case. The likely reason for the insignificant difference between the *small* and *medium* registered state size case is that these effects scale with the number of elements in a state size, and therefore exponentially in the number of state qubits.

Dense Gates

- **General Speedup**:

  Figure 6.6 shows that for relatively small states, up to 7 qubits, the accelerator offers a speedup between factor 3 and 7. For larger states, the quadratic scaling of the accelerator implementation

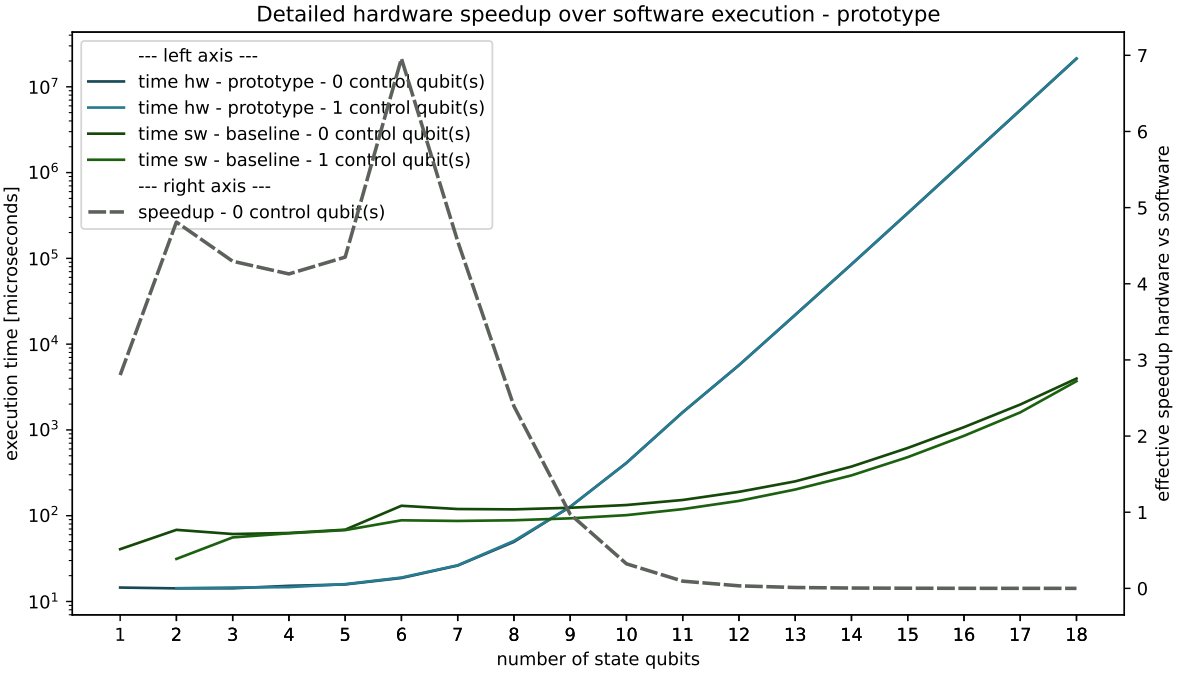


Figure 6.6: Speedups hardware over software for *dense gate* operations

becomes the dominating factor, compared to linear scaling of the software baseline. Therefore, beyond a crossover point at 9 qubits, the current hardware implementation performs worse than the software baseline.

Given a quadratically scaling implementation, and exponential scaling of the problem size, seeing a relatively early crossover point between accelerator and baseline performance is unsurprising. For small states, it is worth noting that the accelerator speedup is actually larger than the speedup for monomial gates in the same range (factor 2-3.5, Figure 6.2). As explained in Section 6.3.2, the main contributor here is the slower baseline computation and matrix assembly, compared to monomial gates. Thanks to element update formulas and pairwise gate operation, the accelerator is not affected by that issue. There is a slight adverse characteristic in the accelerator, which however is insignificant for control overhead-dominated small states, and which is overshadowed by the worse baseline performance. Namely, dense gates in the accelerator are practically guaranteed to be slower than monomial gates on the same state. The host/kernel control overhead is the same, but writing the output vector only starts after reading the entire input vector, leading to additional computation time, even if only one input read iteration is required.

6.3.4. Results - Hardware Profiling

The previous section focused on the differences and tendencies between baseline and accelerator execution time. The analysis here aims at getting a better understanding of the hardware execution in isolation. Figure 6.7 displays the execution times for monomial gates, measurements, and tensor operations, split up into *setup/trigger* phase (setting parameters, activating the kernel), *kernel* (actual execution time in hardware), and *finalize* (detect kernel ready status, query error register). Dense gates were excluded from the plot because they do not add new information in this graph, and the significantly longer execution time for 10 or more qubits would have hindered readability.

The most important observation from the graph is that for small operations (up to 7-10 qubits, depending on the operation type), the execution time, from a software perspective, is completely dominated by the *setup* and *finalization* overhead. This supports the observation from the previous section, where it was determined that the effects related to memory throughput dominate the graph only beyond 10-qubit states. The overhead is more severe for (monomial) gates, than it is for measurements and tensor operations, because gate operations require more parameters - most importantly, the base

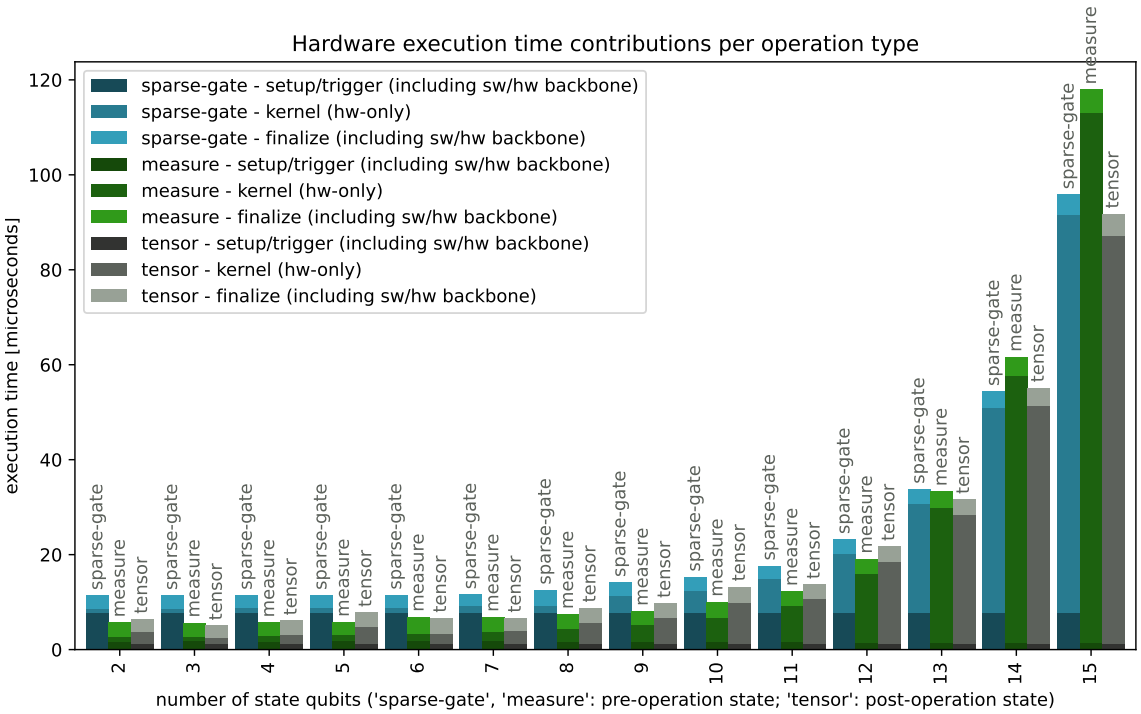


Figure 6.7: Hardware execution profiling for *sparse-gate*, *measure* and *tensor* operations on the prototype. “Setup/trigger” and “finalize” are measured in software, “kernel” is the pure computation time measured in hardware.

operation matrix. Since the parameters are set up via register file, the *setup* phase takes longer. This characteristic contributes to the previously observed higher speedup for tensor operations, for small states, compared to monomial gates. While the same applies to measurements, the more efficient random number generation, and normalization factor computation, are clearly more impactful.

Another perspective on Figure 6.7 is that it reveals the ideal computational speedup potential of the accelerator, without host communication overhead - even for gate operations on small states. Looking at only the hardware without the “instruction interface” might seem unrealistic, but Section 7.3.3 explains that certain types of simulation scenario practically allow eliminating the communication overhead.

For these simulations, the speedup for large states (roughly 12-15 qubits and more) is already known from the previous section, because the contribution of overhead to the total execution time is marginal at these state sizes. Figure 6.7 suggests considerable speedup potential on small states as well. Looking at operations on 3-qubit states, as an example, the data behind the graph looks as follows:

- **Non-Controlled Monomial Gate**:

  accelerator total execution time (with overhead): $11.39\,\mu s$

  software baseline total execution time: $20.08\,\mu s$

  accelerator kernel execution time (without overhead): $850\,ns$

  potential speedup factor (kernel to baseline): 26.62
- **Destructive Measurement**:

  accelerator total execution time (with overhead): $5.63\,\mu s$

  software baseline total execution time: $120.99\,\mu s$

  accelerator kernel execution time (without overhead): $1\,\mu s$

  potential speedup factor (kernel to baseline): 120.99

These figures indicate drastic speedup possibilities when examining the computation in isolation. Section 7.3.3 describes scenarios and future implementation additions that allow leveraging much more of this potential, than the prototype with the NetSquid front end can offer.

6.3.5. Projections

The accelerator is specifically designed for eventually using as much as possible of the HBM multi-channel interface width. The gathered data allows getting a realistic estimate of the accelerator performance, with a fully scaled memory interface. Figures 6.8a to 6.9 show the extrapolated datasets for monomial gates, measurements, and tensor operations, that were previously discussed in Section 6.3.3. Dense gates were omitted, because optimization factors of 8 or even 16 are irrelevant, compared to the quadratic algorithm complexity. Their large-state scaling is discussed in Section 7.2.5 and in Section 9.1.

Data Scaling

In addition to the baseline and prototype performance, the extrapolation yielded two new accelerator data lines. These model a design scaled up to either one full HBM stack (8 times wider than the prototype), or fully scaled up to both HBM stacks, resulting in a 16 times wider interface. The data lines were generated by keeping *setup/trigger* and *finalize* times constant (see Figure 6.7), and scaling the *kernel* execution. Although this creates inaccuracies for small states, it gives a good estimate of the achievable overall performance.

The inaccuracies concern small state scaling, and non-scaling hardware execution cycles. Small states theoretically do not benefit, because below a certain state size, there is not enough data to distribute across 8/16 memory channels. Additionally, the accelerator execution itself contains non-scaling portions, like parameter fetching and memory allocation, in addition to the static steps in the measurement module (determining measurement outcome and normalization factor). However, none of these effects plays a significant role for state sizes beyond roughly 13 qubits. In addition, Section 6.3.4 highlighted that execution times for at least up to 9-qubit states are dominated by host communication overhead, which makes a slightly inaccurate scaling prediction for the kernel execution portion even more tolerable.

As an additional imprecision in the data scaling, it needs to be stated that the impact of memory interface channel congestion are not considered. This makes the data scaling relatively conservative. As Section 7.1.2 explains, an 8-channel or 16-channel implementation allows to generally use memory channels unidirectionally, per operation, because “16-channel” here refers to two distinct sets of 16 channels, one for reading, one for writing. The prototype’s single HBM interface channel in contrast can not avoid operating bidirectionally. Bidirectional operation forces the DDR memory controller to perform constant bus turnarounds, which reduces its throughput. Therefore, the actual throughput scaling factor when going from the prototype to a 16-channel system, for instance, is likely to be greater than 16. The reason to not consider this effect in this section lay in a lack of empirical data on HBM on Alveo platforms, at the time of writing, such that the impact could not be quantified in a dependable manner.

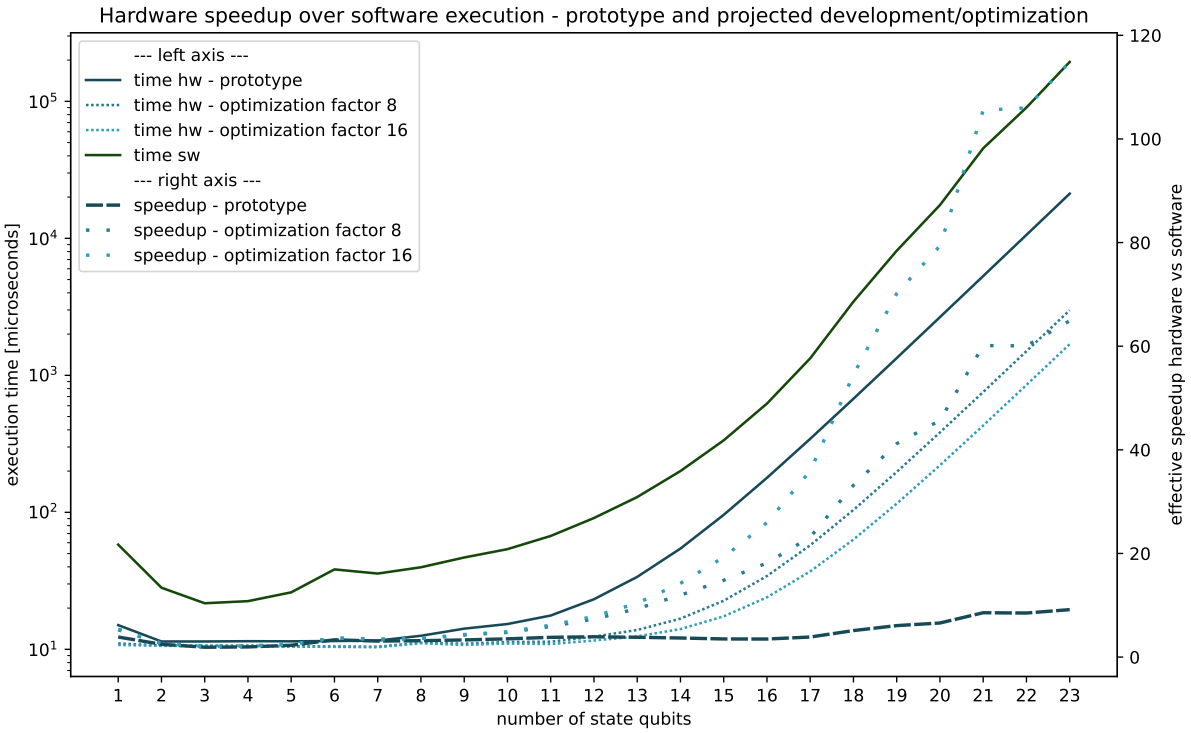
Analysis

Figures 6.8a to 6.9 indicate that, for states beyond 22 qubits, the current accelerator design, with a fully scaled up 16-channel memory interface, can offer speedup factors over the software baseline of at least 110 for monomial gates, 160 for measurements, and 60 for tensor operations. Additionally, one needs to keep in mind that these figures are based on a system running on approximately half of the target clock frequency, while the timing analysis and existing implementations pose no evidence that doubling the current clock frequency is not feasible. It is realistic to assume that the throughput, and thus the speedup, essentially scales with the clock frequency, up to effects like constant memory access latency - whose impact is minimized by sequential accesses of maximum transaction size.

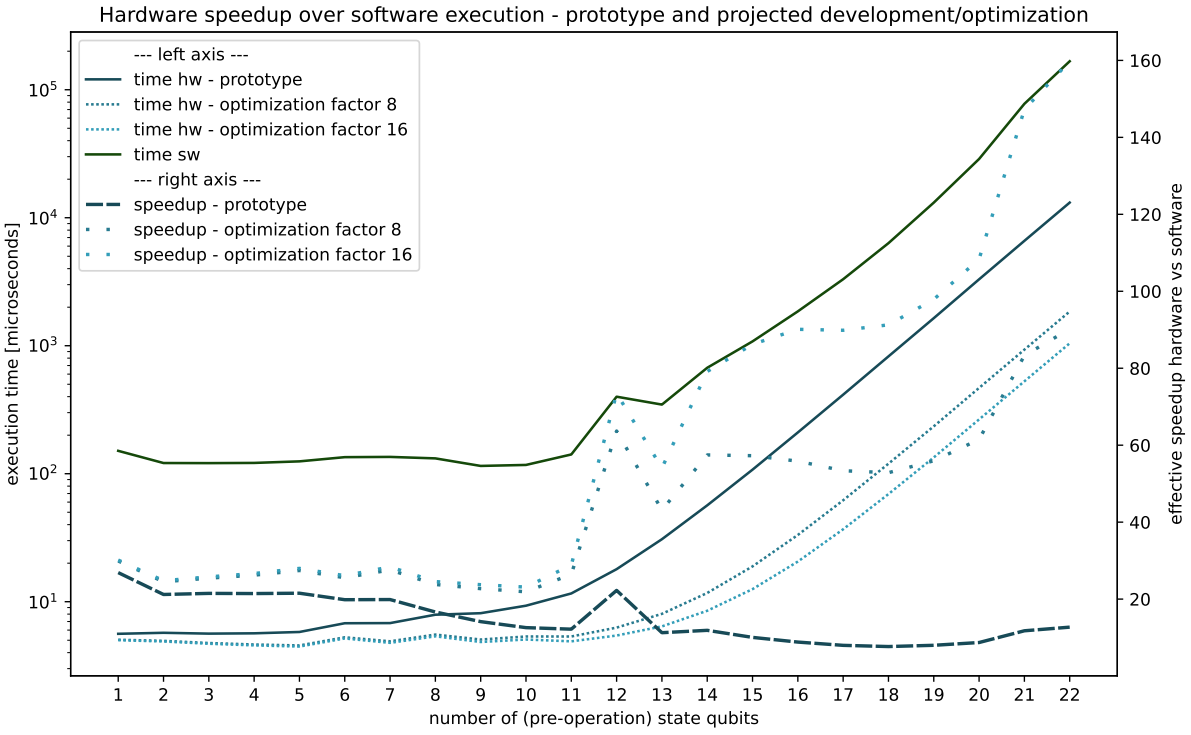
6.3.6. NetSquid Acceleration Potential

The computation performance evaluation is concluded with a study of the acceleration potential when using NetSquid as a front end, in the way proposed in Section 5.8.3. The discussion encompasses the accelerated portion in NetSquid simulations, puts the previous findings from this section into perspective, and finally treats simulation scenarios.

In order to determine the portion that can be accelerated, this section relies on profiling data from the original NetSquid publication [13], displayed in Figure 6.10. In that dataset, the only modules that are considered to be affected by acceleration are *numpy*, and partially *qubits*. In combination, that leads to an accelerated portion of roughly $30\,\%$. Therefore, in this scenario, the maximum achievable speedup factor is $\frac{1}{0.7} \approx 1.43$, which is insignificant in a majority of situations.



(a) Monomial gates. Extrapolation from averaging data for 0 and 1 control qubits (2 control qubits not considered due to rare appearance).



(b) Measurements. Extrapolation from destructive measurement only (because it is the standard case over non-destructive measurement).

Figure 6.8: Speedup projections for memory interface scaling factors 8 and 16, representing one or two HBM stack(s). Raw data filtered as indicated per individual caption.

The most important characteristic of the discussed NetSquid dataset is that it was obtained simulating

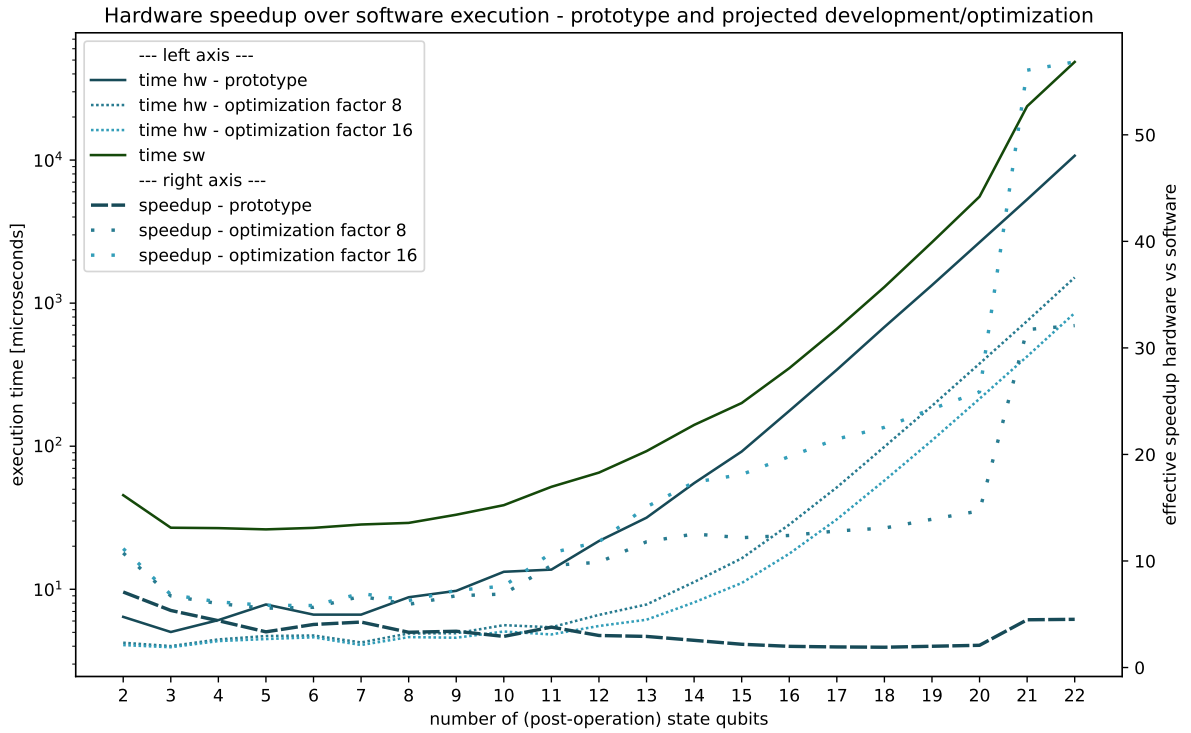


Figure 6.9: Speedup projections for tensor operations, with memory interface scaling factors 8 and 16, representing one or two HBM stack(s). Raw data unfiltered.

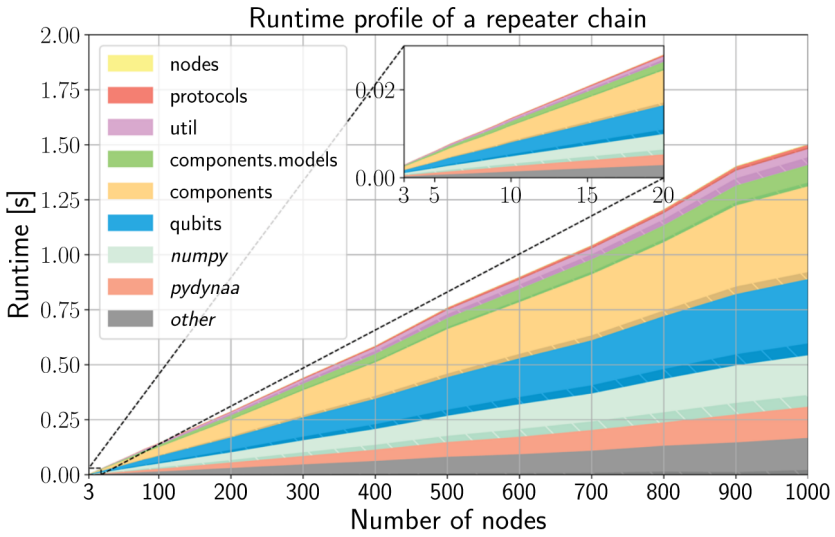


Figure 6.10: from [13]; Profiling data of a NetSquid quantum repeater chain simulation (maximum state size 4 qubits), with varying number of nodes.

one fixed algorithm (a repeater chain), which in this simulation has a maximum state size of 4 qubits. This fact motivates two opposing thoughts: On the one hand, as Section 6.3.3 confirmed, the pure quantum operation computation time scales exponentially in the number of qubits per state. Therefore, simulation scenarios that involve significantly larger states might still greatly benefit from acceleration, if the remaining simulator portions are considered constant (in this case, mostly handling simulated network nodes and event queues). On the other hand, a repeater chain is an extremely common algorithm in Quantum Networking. Other typical techniques in Quantum Networking, like entanglement distillation, also do not usually involve large quantum states. If one assumes scenarios where NetSquid

is used in its original sense, as a Quantum Network simulator, one can summarize these considerations as follows: Accelerating quantum state operations does not benefit many usecases of NetSquid in particular, but select usecases can still see a significant improvement. An example of a such usecase are studies in the direction of surface code-based Quantum Networking ([41, 42]). Section 7.3.4 provides a theoretical path towards a meaningful acceleration for currently non-benefitting simulations like repeater chains.

Analogous to the "general" speedup discussion, which ended with an extrapolated projection in Section 6.3.5, the elaboration on NetSquid is concluded with a projection on the potential speedup for operations relying on large quantum states. For that, the computation time data from Figure 6.10 is scaled up from 4 qubits (although not all operations are on 4-qubit states) to 22 qubits, based on this work's measurements. Afterwards, the portion that can be accelerated, in a respective NetSquid simulation, is estimated, by assuming the same non-computation time as the repeater chain, but the scaled up computation time. Lastly, the overall acceleration factor is determined by accelerating the scaled up computation time with the projected factors from Section 6.3.5:

- **from 4 to 22 qubits**:

  gate operations factor 4558, measurements factor 1375, tensor operations factor 1846

  assuming an average share of 1 tensor operation, 2 gates, and 2 measurements, the averaged execution time increases by factor 2742.4.
- **accelerated portion**:

$$
\begin{aligned}
p = 0.3,\ \ t_p &= p \cdot t \\
t^* &= 0.7 \cdot (t - t_p) + 2742.4 \cdot t_p \\
p^* = \frac{t_p^*}{t^*} &= \frac{2742.4 \cdot t_p}{0.7 \cdot (t - t_p) + 2742.4 \cdot t_p} \\
&= \frac{2742.4 \cdot t_p}{\frac{0.7^2}{0.3} \cdot t_p + 2742.4 \cdot t_p} \\
&\approx 0.9994 \equiv 99.94\,\%
\end{aligned}
$$

- **acceleration factor**:

  again assuming on average 1 tensor operation (speedup factor 60), 1 monomial gate (speedup factor 110), 1 dense gate (speedup factor 1, assumption for either a BRAM-extended dense gate array, proposed in Section 7.2.5, or the alternative random memory access scheme outlined in Section 9.1), and 2 measurements (speedup factor 160), therefore an overall speedup factor of $s = 98.2$.

  Applying Amdahl's law yields:

$$S = \frac{1}{1 - p^* + \frac{p^*}{s}} = \frac{1}{1 - 0.9994 + \frac{0.9994}{98.2}} \approx 92.79$$

Concluding, as an extremely broad estimate, the proposed accelerator could make a Quantum Network simulation with an "average state size" of 22 qubits faster by a factor of 92.79, if using the full bandwidth of one HBM stack. As indicated during the above calculation, this estimate is subject to finding a large-state dense-gate implementation that at least performs equally good as the software baseline.

The speedup increases further, if in addition the clock frequency can be increased to the intended $450\,\mathrm{MHz}$ (Section 7.1.1), and if as expected the throughput scales almost linearly with the clock frequency for large states. Assuming an overall speedup factor of $s = 180$, to compensate for non-ideal scaling, yields

$$S = \frac{1}{1 - p^* + \frac{p^*}{s}} = \frac{1}{1 - 0.9994 + \frac{0.9994}{180}} \approx 162.54$$

6.3.7. GPU Acceleration Potential

The previous sections discussed speedup projections of the FPGA prototype, both standalone and integrated into NetSquid. Earlier in this thesis, Section 4.5 elaborated on the feasibility of porting the prototype accelerator design to a GPU. Since the results were relatively promising, this section provides projections on the expected performance of a such implementation. In contrast to the previous sections, this study is carried out qualitatively, but not quantitatively.

Elaboration

Similar to the performance analysis in Section 6.3.3, different effects are relevant, depending on the average state size in quantum operations. Accordingly, the following discussion treats small and large states separately:

- Small States
  - As demonstrated in Section 6.3.4, for small states, communication latency between host and accelerator accounts for the majority of the computation time. Therefore, the most impactful factor is the ratio between FPGA and GPU host-accelerator communication latency. Although conducting a data-based study was beyond the scope of this thesis, it is expected that this latency for a GPU is slightly lower, due to more hard-IP components, and a higher degree of optimization.
  - The performance results for quantum measurements in Section 6.3.3 revealed that the FPGA is particularly effective at mathematical operations where it can rely on dedicated circuits, while the CPU has to revert to generic instructions - namely random number generation, and inverse square-root computation. On the one hand, as highlighted in Section 4.5.1, these operations are implemented for execution on GPU, such that one does not have to fall back to the host CPU. On the other hand, since the GPU does not have dedicated circuits either, the FPGA is expected to maintain its advantage.
  - The FPGA implementation incorporates efficient on-chip memory management, while for the GPU, the host CPU needs to take over that task. For small states, with short overall computation times, it is expected that memory management on the host CPU causes noticeable latency, compared to the FPGA system.
  - The FPGA offers full control over local on-chip memory, which can be used as quantum state memory, discussed later in Section 7.2.4. The GPU can only do that “indirectly” (considering globally accessible memory), by relying on caching mechanisms. This gives the FPGA deterministic, faster accesses to states in on-chip quantum state memory.
  - As will be discussed later in this thesis, in Section 7.3.3, there are realistic scenarios which allow the accelerator to handle a quantum instruction queue, eliminating any host-accelerator communication. However, only the FPGA offers the resources to implement this type of autonomous simulation, while the GPU relies on instruction kernels being issued by the host CPU. Eliminating the host-accelerator communication would put the FPGA at a dramatic advantage, in these scenarios.
- Large States
  - The FPGA system is designed and projected to eventually leverage the full available HBM bandwidth. In order to compete, the GPU therefore needs to be equipped with comparable or even more capable memory, and be able to saturate that bandwidth, in terms of computational throughput. It is estimated that, due to the relatively simple kernels per CU, computational throughput is unlikely to turn out as the limiting factor. Therefore, the eventual speedup estimate depends on the specific GPU model, and most importantly its memory bandwidth.
  - Dense gate computation, which is the bottleneck for the FPGA system, generally experiences the same schematical issues on a GPU. However, several characteristics are expected to deviate. First, considering array-based dense gate computation (as explained in Section 4.3.3, and improved in Section 7.2.5), it comes down to the specific GPU model and architecture, how large of an array can be realized, and how efficient the data cascading can be performed. Random-Access computation, as outlined for large states later in Section 9.1, is expected to perform better than on FPGAs, due to integrated cache memory, which is automatically handled.

Summary
Based on the above elaboration, it is projected that, in simulations on small states, a GPU implementation needs to rely on faster host communication to compete with an FPGA system, but is likely to have a slightly higher speedup potential when dealing with large states. For small states, the FPGA is expected to gain an advantage through on-chip memory management, more efficient local on-chip quantum state memory, and dedicated circuits for auxiliary mathematical operations. The GPU's only benefit is a potentially lower host-communication latency, which a future study needs to quantify. Additionally, the ability to implement on-chip instruction queue handling raises the FPGA's speedup ceiling to a level that is not achievable by a GPU. For large states, the elaboration yielded no clear estimate, but rather demonstrated that the particular GPU model and architecture play a significant role. However, since there are GPUs with competitive or higher memory bandwidths to the U55C's HBM, and due to better (automated) caching, it is considered realistic that, for states in the range of 25 and more qubits, GPUs offer a higher performance ceiling than FPGAs.

6.4. Auxiliary Results

After the previous section presented results on actual quantum state vector operations, this section focuses on additional findings, that do not contain full accelerator operations. First, Section 6.4.1 highlights how the internal number representation and DSP port assignment were derived for the on-chip inverse square-root implementation, presents accuracy data for the algorithm in isolation, and sketches future improvement steps. The second part, Section 6.4.2, presents findings on host-kernel communication latency with Alveo accelerator cards and XRT host software, for both register file and shared memory interaction. These findings led to the eventually implemented method of operation parameterization, and can prove useful for potential future project iterations, as laid out in Section 7.3.3.

6.4.1. Goldschmidt-Algorithm on (Ultrascale+) DSPs

As indicated in Section 5.5.4, there are several degrees of freedom when tuning the DSP-based Goldschmidt inverse square-root algorithm. These concern the assignment of signals to multiplier inputs (in other words, which signal to truncate to 18 bits, instead of 27), and the fixed-point number representation of the result. The input uses the system-wide representation of 1 sign bit, 1 integer bit, and 25 fractional bits, which was designed for a number range of $[-1, 1]$. In the normalization factor computation during measurements, the inputs to the inverse square-root function are measurement expectation values, and thus positive numbers in the range $[0, 1]$. The resulting output value range is $[1, \infty)$. Accordingly, a different number of integer bits is required for the result, with a suitable balance between extended number range via more integer bits, and sufficient fractional bit precision for numbers close to 1.

The best-performing configuration, in terms of DSP port assignment and fixed-point parameterization, was determined empirically, by modeling the circuit from Figures 5.12 to 5.13 with python's `fxpmath` package. As described in Section 5.5.4, the design space was slightly reduced for the prototype, by keeping the variable r the narrow DSP input in all cases, and by defining only two classes of fixed-point number representation. Thus, the experiments were restricted to either making h or x the wide input of the upper DSP (Figure 5.12), and to determining the number of integer bits for the representation of h, and of the result. The most relevant datapoints that were obtained during these experiments are displayed in Table 6.3. The exact test modalities are described in the figure caption, and will not be further elaborated in the text. All experiments use the initialization scheme displayed in Figure 5.14, which was also modeled in python.

The data in Table 6.3 led to the conclusion of using h as the wide multiplier input, and employing 13 integer bits for the result (and for h), by the following reasoning: Below 10 integer bits, the *small*-range accuracy for both port assignments becomes unacceptable. *wide x* is only usable at exactly 10 integer bits, because above that figure, the *full-range* accuracy deteriorates. However, with a *wide x*/10 integer bits configuration, the *small*-range pass rate of $94\,\%$ is acceptable, but not good, and the worst-case relative delta of 2.016 is rather high. *wide h* configurations offer $100\,\%$ *full-range* accuracy for higher numbers of integer bits, and manage to combine this with around $96\,\%$ pass rate for *small* numbers, at 11-14 integer bits. Among this range, 13 integer bits has the best worst-case delta at 0.067, which is also significantly better than the best *wide x* datapoint.

The investigation into the Goldschmidt-Algorithm is concluded with pointing to potential avenues

result int bits	iterations	wide h			wide x		
		small		full-range	small		full-range
integer		rel. Δ	pass	pass	rel. Δ	pass	pass
9	5	2.007	$72.2\,\%$	$100\,\%$	2.011	$72.7\,\%$	$100\,\%$
10	5	2.016	$92.1\,\%$	$100\,\%$	2.016	$94\,\%$	$100\,\%$
11	5	2.040	$96.6\,\%$	$100\,\%$	2.040	$97.3\,\%$	$86\,\%$
12	5	2.167	$95.6\,\%$	$100\,\%$	0.498	$97.9\,\%$	$59.3\,\%$
13	5	0.067	$96\,\%$	$100\,\%$	0.135	$97.3\,\%$	
14	5	0.418	$96\,\%$	$100\,\%$	0.270	$98.1\,\%$	

Table 6.3: Accuracy for various simulated parameterizations of the Goldschmidt inverse square-root algorithm implementation. *result int bits*: integer bits in the result/h fixed-point representation (fractional bits for h are determined by wide h/x); *iterations*: algorithm iterations; *wide x/h*: which signal is connected to the wide first-level multiplier input; *small/full-range*: small numbers are randomly, uniformly drawn from range $[2^{-25}, 2^{-15}]$, large numbers are drawn in the same way from range $[2^{-15}, 1]$; *rel. Δ*: worst-case relative deviation from correct value; *pass*: rel. $\Delta < 1 \times 10^{-2}$; blank datapoints were not obtained due to irrelevance

for future accuracy improvements. First and foremost, the restrictions to the design space in terms of fixed-point representation classes and port assignment can be relaxed. An improvement can be expected from employing 1 more integer bit for the result, than for h (because the result is obtained by computing $h \cdot 2$). The representation categories were roughly determined by orders of magnitude in powers of 10, not in powers of 2. Considering DSP port assignment, it is not a given that making r the narrow input of both of the second-stage DSPs is ideal. It only proved to be "good enough" for a functional prototype, such that no alternatives were studied. Lastly, the DSP48E2 primitive natively allows assembling wider multipliers, by connecting two adjacent instances via built-in carry chains and bit-shifted cascading lines. This way, multiplicand widths can be increased from 27x18 bits to 45x45 bits, for instance. This way, none of the inputs would have to be "truncated" to fit an $18\,\mathrm{bit}$ multiplier port, and internally, during the algorithm iterations, the circuit could operate at an even higher precision - before truncating the result to $27\,\mathrm{bit}$, for the following post-measurement state normalization DSP array in the engine's measurement module. The hardware cost is negligible, because the inverse square-root module is instantiated only once across the entire kernel.

6.4.2. Alveo Kernel Data Exchange Latency

As mentioned in Section 5.7.1, two different ways were investigated for passing operation parameters from host to accelerator. One approach consisted in using one bulk write to shared memory, via host-side DMA. The other, eventually faster approach, consisted in separate writes to additional registers in the register file. This section presents the execution times of both techniques, for future reference. It is crucial to note that all data was obtained using HBM as shared physical memory, the effects of which will be discussed at the end of this section.

Experiments showed the following access latencies:

- **Shared HBM Memory** ($1\,\mathrm{kbit}$ DMA transmission, 1 register file kernel trigger): $25\,\mu\mathrm{s}$ to $65\,\mu\mathrm{s}$ write value range across experiments, while values were relatively stable within one experiment
- **Register File** (single $32\,\mathrm{bit}$ register access): $0.104\,\mu\mathrm{s}$ write, $1.177\,\mu\mathrm{s}$ read

  average access times through $1\,500\,000$ total accesses, iterating through the entire register file

Even for gate operations, which require the most parameterization with $92\,\mathrm{B}$ or 23 register file writes, it is evident that passing parameters is still drastically more efficient via register file, than via shared HBM memory. The other way around, an operation would have to require roughly 400 register file accesses, or about $1.6\,\mathrm{kB}$ of parameterization data, in order for shared memory to become a useful alternative.

It is expected that lower-latency shared memory can reduce the disadvantage to an extent, but not significantly. The Vitis environment supports setting up DMA access to BRAM and UltraRAM as well, which both have a considerably lower access latency than HBM. However, although it has not been validated in conjunction with DMA, the HBM documentation [37] suggests that the majority of the latency

lies in the general DMA accessing, and not in the memory activation latency: The document states that the HBM memory controller has a minimum latency of 128 cycles, with global addressing enabled, at a closed-page access (worst-case scenario). Even with a conservative estimate of 300 cycles access latency, at the current clock frequency of $224\,\mathrm{MHz}$, the HBM activation itself would only take $1.339\,\mu\mathrm{s}$. This would account for not more than $2\,\%$ to $5.36\,\%$ of the total software-measured operation setup time, such that a lower-latency physical memory would only have a negligible impact on the total parameterization time.

6.5. Conclusion

This chapter presented any relevant findings concerning the implemented prototype. The prototype was analyzed in terms of hardware resource usage, correctness and reliability, and computation performance. Additionally, quantitative findings on components that may also appear in unrelated FPGA usage scenarios are discussed separately.

Section 6.1 studied the implementation figures of the current design. The kernel clock frequency was projected to approach the intended $450\,\mathrm{MHz}$, although it is currently limited to roughly half that, mostly due to two foreseeable critical paths. Both have relatively straightforward solutions, hence the optimistic projection. Additionally, the chip resource usage of the current kernel and the Vitis shell indicate that the kernel is scalable to substantially wider memory interfaces, as intended. The current kernel's hardware footprint is almost negligible, and while the Vitis shell itself consumes a considerable amount of resources - expected to increase with a wider shell-managed memory interface - the impact is still not expected to limit kernel extension. The estimate is supported by the fact that the Vitis shell uses up close to no DSPs, which leaves these available for the kernel's computation units.

Section 6.2 discussed validation and reliability testing of the hardware implementation. Correct functionality was ensured for operations on states of up to 22 qubits. The $27\,\mathrm{bit}$ fixed-point representation was found to cause no meaningful accuracy loss, compared to the $64\,\mathrm{bit}$ floating-point baseline. In terms of computation accuracy however, it was concluded that the on-chip inverse square-root computation - while yielding acceptable results for the prototype phase - requires further tuning. The entropy source for measurement outcomes was found to operate correctly, although the data indicates that a shorter LFSR could reduce slight temporary measurement outcome biases.

Section 6.3 studied the per-operation performance of the accelerator, against a software baseline on an otherwise idle HPC CPU node (single-thread), on varying quantum state sizes. Next to the pure execution time, the accelerator was profiled to gain insights on communication latencies. Additionally, projections were extrapolated for the performance of a future implementation, which would use the full HBM interface width. All operations yielded a speedup between factor 2 and 5 for small states (up to 5 qubits). Dense gate operations saw a crossover point between hardware and software performance at 8 qubit-states, due to the quadratic scaling of the current hardware implementation. All other operations experienced a significant increase in speedup (to factors 4 to 15), once the state size exceeded the CPU's cache memory capacities, which in the baseline happened at 21 qubits. All speedups include (constant) communication latency. It was found that not having to compile a full operation matrix offers a tremendous performance advantage, even more so with controlled gates than with non-controlled gates.

For small states (up to roughly 6-12 qubits, depending on the operation type), the accelerator execution time was dominated by communication latency, not by kernel computation time. The data additionally revealed that the inverse square-root computation and probabilistic measurement outcome determination in hardware are drastically faster than in software. That effect however does not scale with the state size, and thus becomes insignificant for state sizes beyond 10 qubits.

When taking the application front end into consideration, it was found that accelerating the NetSquid simulator in particular is in many usecases not promising with the current accelerator architecture. The reasons are the non-accelerated NetSquid overhead, and the mostly small state sizes in Quantum Networking scenarios - combined with the dynamic or even unpredictable instruction order, which does not allow an on-chip instruction queue to prevent the communication latency at every operation. However, extrapolations were formulated for the accelerator's speedup potential on large states (22 qubits and more), with a full-width HBM interface, and an alternative dense gate implementation that is on par with the software baseline for large states. In these cases, the pure per-operation speedup factors would range from 60 to 160 (except for dense gates). In a realistic simulation scenario, including

NetSquid overhead, an estimated simulator speedup of almost 93 was extrapolated, which would increase to roughly 162 with the intended $450\,\mathrm{MHz}$ kernel clock frequency.

Lastly, Section 6.4 presented auxiliary results that emerged during the thesis work, and that are generally applicable to FPGA computing applications. Section 6.4.1 focused on inverse square-root computation, describing how careful tuning of fixed-point number representation and port assignment yielded considerable computation accuracy, at an extremely limited hardware footprint (only 3 DSPs), and even at limited degrees of freedom. Additionally, it is projected that the accuracy can still be boosted significantly, at an acceptable increase in DSP usage, and via more fine-grained exploration of degrees of freedom. Section 6.4.2 provided a comparison of host-accelerator communication latencies for AMD Alveo FPGA accelerator cards. It was determined that a single $32\,\mathrm{bit}$ register file host-side write is roughly 250 to 650 times faster than a DMA HBM host write and subsequent kernel read, while documentation suggests that the memory activation accounts for no more than $5\,\%$ of the DMA access time - making the type of FPGA memory that DMA connects to irrelevant in this comparison. Therefore, the crossover point for data transfer from host to kernel is around 450 registers, or $1.6\,\mathrm{kB}$, from a pure performance standpoint.

7

Prototype Optimization and Extension

The previous Chapters 3 to 6 derived and explained the accelerator prototype's current status, in terms of state vector representation and computation scheme, high-level design, and FPGA implementation. This chapter is dedicated to planned improvements and further development iterations, based on the delivered prototype. In contrast to future work topics discussed later in Chapter 9, this chapter contains only optimizations whose impact on performance, usage scenarios and implementation has already been studied. Chapter 9 is reserved for potential extensions that have not been substantiated yet through more in-depth exploration. Every approach in this chapter is first explained functionality-wise. The level of detail ranges up to ready-to-implement designs, mostly in the earlier sections.

Due to the high number of improvements and approaches, they have been divided into categories for this chapter. The following sections each treat one category, ordered by the "amount by which they alter the current prototype's low-level architecture or functionality set". The first group, discussed in Section 7.1, recapitulates and solidifies improvements that have already been elaborated during this thesis. Afterwards, Section 7.2 presents novel optimizations within the present hardware architecture, mostly to individual design units. Lastly, Section 7.3 discusses substantial functionality additions, that either directly or indirectly affect the (host application) software level.

7.1. Suggested Improvements

In Chapter 6, several improvements of the already-existing accelerator have been discussed in detail, partially with quantitative projections on the system performance. These concern system clock frequency, HBM memory interface width, and accuracy of the inverse square-root circuit in the engine's measurement module. The following sections reiterated already formulated findings, and add necessary depth for realization in hardware.

7.1.1. Clock Frequency Optimization

Section 6.1 highlighted that the currently achieved accelerator system clock frequency is $224\,\mathrm{MHz}$, while the eventual target is $450\,\mathrm{MHz}$, to maximize HBM interface throughput. In addition, it was explained that the most problematic timing paths lie in the dense gate engine module index filtering, and in the floating-point to fixed-point converters. Both of these do not pose a substantial problem, since they are both low-fanout local fabric logic paths, which allows for an additional timing register. The impact on computation performance is negligible, because one extra pipelining stage means adding $2 - 5\,\mathrm{ns}$ to a computation time in the range of $10 - 15\,\mu\mathrm{s}$, or roughly $0.2 - 0.3\,\%$. The long-term projection, as explained in Section 6.1, is that $450\,\mathrm{MHz}$ is a realistic target frequency, and that, as a result, the pure kernel computation performance (which dominates the computation time for states of about 12 qubits and more) can almost be doubled.

7.1.2. Full-Width HBM Interface

In Section 6.3.5, accelerator performance projections were formulated for a system that operates more than the currently used 2 HBM channels. It is important to point out that, in order to enable operating the HBM at maximum bandwidth, one needs to not only occupy all channels, but also deactivate the

integrated hard-IP crossbar, and thus global addressing. Deactivating global addressing however requires substantial changes to the quantum state memory module's execution block. For this reason, this chapter proposes two designs: One final design, with the crossbar deactivated, and one intermediary design, with activated crossbar, that is based on the current operation scheme. This section contains the intermediary design, which lines up with the projections presented in Section 6.3.5. The final proposal is explained later in this chapter, in Section 7.2.3, because it requires more substantial changes to the quantum state memory module's data execution block (module naming see Figure 4.7).

The main challenge when using multiple AXI channels for quantum state memory lies in keeping all of these channels occupied, as much as possible. It is therefore imperative to efficiently distribute quantum states in memory across the HBM's PCs, or MCs (naming consistent with Xilinx documentation). As a visualization, if a state is entirely located in one PC, accessing it from 4 parallel AXI channels will not increase throughput, because all accesses are handled by the same MC, which is designed to saturate exactly one AXI channel. In practice, the effective throughput likely even degrades, because the MC needs to "jump" between accesses to serve, and thus memory locations, instead of performing one contiguous operation.

The intuitive solution to data distribution across PCs would be to simply split up the state into N batches (with N being the number of memory channels used), and assign one batch to each PC. The following elaboration shows that this method is a good starting point, but needs further tuning, because in this form, it has some detrimental effects for the achievable throughput.

First, an advantage is that distributing data equally across PCs makes memory allocation relatively simple: For the memory allocator, it suffices to only look at one PC, and manage allocations for the data batches in that channel. The memory locations for the other PCs can be kept in sync with the first channel. States that are too small to occupy at least one memory page in each channel can be "padded", to keep the allocation pattern the same across all channels.

The flaw of the formulated solution lies in memory access congestion: Almost every quantum state operation requires reading from one location, and writing to another location, at the same time. Ideally, the hardware that handles reading and writing is completely separated from each other. Otherwise, physical MCs have to constantly alternate between read and write mode, and between memory locations. With states distributed across all PCs, that type of congestion would be guaranteed to happen, at every operation. Therefore, states need to be distributed across PCs in a way that prevents congestions, as much as possible.

When defining an efficient memory allocation scheme, a tradeoff needs to be found between preventing read/write congestion within channels, and keeping memory management complexity at a minimum. The proposed solution is to divide the PCs into two virtual groups, named A and B in this section. Ideally, during an operation, a state would be read from group A, and written back to group B, or vice versa. AXI channels alternate between read-only and write-only operation (channel 0 is read-only, channel 1 write-only, etc.), to ensure that AXI channels are at least "close" to the data they access, which reduces the crossbar-induced latency. PCs are assigned to groups in pairs, because every MC operates two PCs in sync (PC 0,1,4,5,... in group A, PC 2,3,6,7,... in group B). Consequently, the memory allocator needs to look at 2 PCs (one per virtual group), instead of one, which is still manageable. Since the memory allocator does not know operation semantics, it can not tell with certainty whether a certain allocation request should result in a specific PC group, and if so, in which one. The heuristic proposal is that, whenever updating an existing state allocation in one group, the memory allocator prioritizes an allocation in the other group. An allocation update usually happens when a control block FSM (see Figure 4.7) requests a memory destination for a post-operation state, such that during the operation, source and destination reside in different PC groups. In addition, it is advisable for the allocator to maintain an overview of overall memory occupation per PC group, in order to keep the amount of occupied memory balanced across both groups, for instance when creating a new state.

The proposed solution does not pose any substantial difficulties in terms of hardware implementation. Since components like the engine data bus are designed to operate with a multi-channel memory interface, increasing the number of channels does not unnecessarily inflate any dependencies, or cause scaling problems. Several units might still need some adaptation, like the engine dense gate module, and the engine return data path throttling buffer.

Firstly, the dense gate computation array's per-level adder is currently realized as a multi-operand addition, in fabric logic (see Section 5.5.3). While this method is sufficient for a 4-lane databus, at roughly half the target clock frequency, it is practically guaranteed to fail timing at $450\,\mathrm{MHz}$, and 16 memory

channels/64 lanes. A solution would be the DSP-based accumulation tree that is already utilized in the engine measurement module (Section 5.5.4), together with the respective latency-adjustment for the output engine data bus control signals.

The engine return data path buffer, in its current form, has 2 potential problems. First, it might simply prove too small to efficiently buffer the much larger throughput of an 8 or 16 times wider memory interface, causing more stalls in the system. Additionally, it is advisable to employ multiple parallel buffers - one per memory interface - instead of one large buffer. With one large buffer, a dataword can only be taken from the FIFO if **all** memory interfaces are ready to receive data. Therefore, non-synchronous availability across memory interfaces to receive data, for instance due to physical DDR effects, can cause the data buffer to fill up unnecessarily. Separate buffers can handle dynamic availability significantly better - while generally still all reading must be halted, if only one buffer reaches its fill threshold. Additionally, one single buffer creates a control dependency across all memory channels, because all their `ready` signals need to be AND'ed - which is not a severe issue, but beneficial to avoid.

Since the engine scales with the engine data bus width, it scales directly with the memory interface width. Therefore, an 8 or 16 times wider engine data bus effectively translates into an 8/16 times increase in DSP usage. Section 6.1 pointed out that the current kernel implementation fits entirely into SLR 0, in the U55C's FPGA. The projection is that this scaled-up engine itself, with the current dense gate implementation, still fits into SLR 0. It consumes 117 DSPs, at the time of delivery, which is $4\,\%$ of the available DSPs in SLR 0, such that even an increase by factor 16 is within SLR 0 boundaries.

Nevertheless, even if the scaled-up engine itself does not necessitate SLR crossings, the inferred on-chip vitis shell might still do so. That grows as well, because it provides the axi interfacing with the physical HBM. The share of the memory interfacing in the total shell resource consumption was not determined. However, given that the shell currently occupies $18.74\,\%$ of the CLBs in SLR 0, and assuming a share of $20\,\%$ memory interfacing in that, the shell would jump to using up almost $80\,\%$ of SLR 0's CLBs - in which case it is likely that adding the kernel requires SLR crossings.

7.1.3. Inverse Square-Root Accuracy

Inverse Square-Root accuracy refers to the Goldschmidt-Algorithm implementation in the engine's measurement operation module. Section 6.4.1 explains that the current implementation was tuned to an extent that provides sufficient accuracy for a proof-of-concept, but not necessarily to the maximum achievable accuracy yet. As the section furthermore explains, several degrees of freedom were temporarily removed, and the DSP multiplier accuracy can be increased significantly by doubling the number of DSPs.

Optimizing the existing inverse square-root implementation is a relatively uncomplicated process. The additional degrees of freedom can conveniently be integrated and evaluated using the already-existing python simulation. The best way of increasing the DSP accuracy can be studied as well (the options exist to either combine two DSPs into a 45x45 multiplier, or into a 54x36 multiplier). The effects on the hardware implementation are marginal. The fine-tuning in terms of bitwidth equates to pure bitstring interpretation, the design leaves the inter-DSP native routing available for any of the wide multiplier options, and the additional 3 DSPs are insignificant, even compared to the remainder of the engine's measurement module alone.

7.2. Further Prototype Optimization

This section discusses proposed optimizations to existing design units, within the current architecture. In contrast to the previous section, it is dedicated to aspects that have not yet been explicitly mentioned or elaborated during the thesis.

7.2.1. Memory Interface AXI Width Adaptation

This paragraph aims to correct an imbalance in engine throughput and memory interface throughput, caused by the generic Vitis shell-managed $512\,\mathrm{bit}$ AXI memory interface. The Vitis shell provides $512\,\mathrm{bit}$ wide AXI interfaces for any physical memory that it manages, regardless of the actual memory type. The width of the parallelized engine operation cores is determined by the engine data bus width, which in turn depends on the memory interface. Consequently, since vector elements are aligned to $128\,\mathrm{bit}$ boundaries in memory, the current configuration creates a 4-lane engine data bus (per direction), and engine operation modules with 4 parallel CUs. The HBM however natively provides a $256\,\mathrm{bit}$ AXI

interface, at a maximum frequency of $450\,\mathrm{MHz}$ [37]. Therefore, with the current ratio between engine operation module width and memory interface, it is impossible for the physical memory to saturate the potential engine throughput. An AXI bus converter, that converts the $512\,\mathrm{bit}$-wide Vitis-generated interface back to $256\,\mathrm{bit}$, is the proposed solution. This would halve any data path-related FPGA resource consumption in the accelerator, while theoretically not costing any throughput. A possible alternative would be to keep $512\,\mathrm{bit}$ AXI interfaces to the kernel, run only the physical memory interface at its maximum $450\,\mathrm{MHz}$, and restrict the engine to half the memory clock frequency. This solution however costs substantial extra hardware, and is only advisable if the first solution is not technically feasible (for instance in terms of clock frequency), for which there currently is no indication. Accordingly, the accelerator's HBM memory interface should be converted back to $256\,\mathrm{bit}$ datawords.

7.2.2. Low-latency Memory Allocation

These paragraphs address the latency of the quantum state memory module's memory allocator (see Figure 4.7). Determining a new memory location for a quantum state is part of practically every quantum state operation, namely when the control block FSMs (Section 4.4.5) request a target memory location for the post-operation state. In most cases, the associated latency fully contributes to the total operation execution time, because there is no task to carry out in parallel for the FSMs, while waiting for the requested allocation data.

These latency contributions can be significant, when looking at operations on relatively small states. In the prototype, a memory allocation takes at least 87 cycles (3 cycles to check 1 allocation table entry, table size of 29), or about $383\,\mathrm{ns}$. This number can grow with the number of states in the system, such that $500\,\mathrm{ns}$ is a realistic figure for the experiments conducted in this thesis (Section 6.3.3). Considering pure kernel execution time, a monomial gate, as an example, takes about $850\,\mathrm{ns}$ on a 1-qubit state, and $1.5\,\mu\mathrm{s}$ on a 9-qubit state. Therefore, for these state sizes, memory allocation can account for $30\,\%$ to $50\,\%$ of the kernel execution time.

The memory allocation latency is certainly not critical in the benchmarking scenario in Section 6.3.3, because even 1-qubit monomial gates have a total average execution time of more than $11\,\mu\mathrm{s}$, due to communication overhead. However, there are realistic scenarios when the current memory allocation implementation becomes a bottleneck. First, the latency depends on the memory allocation table size - referring to the maximum number of entries, not the actual number of entries. That number is currently 29, but certain simulation scenarios might require up to 1000 (small) quantum states. In this case, the current implementation would impose a memory allocation latency of more than $13\,\mu\mathrm{s}$, in the best case, and easily more than $100\,\mu\mathrm{s}$ in non-ideal cases. Additionally, future design variations with negligible host-accelerator communication latency (with on-chip instruction queues, for example, see Section 7.3.3 later in this chapter) would experience significant performance losses from the current implementation, due to the high share of the memory allocation latency in total kernel execution time. Two approaches are presented in the following for reducing that latency, compared to the prototype implementation.

Background Pre-Computed Memory Allocation

At the time of delivering this thesis, the memory allocator is implemented to only become active when actively queried by the memory allocation table. Consequentially, the state machine to handle new allocation requests is generally idle while the accelerator performs the actual computation. The proposal is to extend this state machine, such that it can pre-calculate allocations for specific state sizes in the background, instead of idling. The memory allocator module then maintains its own table of valid new memory locations, per state size. Upon a request, it can immediately return the respective pre-computed entry, if available.

There is a high chance that the state machine is generally not able to pre-compute an allocation for any possible state size, before the next allocation request. Therefore, state sizes that are more likely to be requested should be prioritized. While this can not be done with certainty, a good heuristic is to first process state sizes that are already present in the memory allocation table - ordered by the number of appearances, ideally. The reason is that during simulation, allocation requests usually originate from gate operations, which do not change a state's size (with the exception of destructive measurements). Additional favorable state sizes would be 2 qubits (for EPR pairs), and state sizes that are 1 smaller than existing states (for destructive measurements). When combined with a full-width HBM interface with virtual PC groups (as described in Section 7.1.2), the PC groups can easily be taken into consideration for the heuristics - for instance, when pre-computing an 8-qubit allocation, because there already is one

8-qubit state in the allocation table, the speculative allocation should be computed for the opposite PC group than the existing 8-qubit state, to ensure congestion-free dataflow. Predicting tensor operations is more difficult, because it requires looking at pairs of existing state sizes. It is however also less problematic, because the tensor operation control block FSM (Algorithm 8) uses the memory allocation latency to fill the engine tensor module buffer with the static state (see Section 4.3.2).

It is expected that the presented approach of pre-computing allocations in the background, combined with heuristic state size prioritization, can hide an immense part of the memory allocation latency during operation simulation.

Memory-Centric Allocation Algorithm

One potential disadvantage of the existing memory allocator is that its allocation latency depends on the maximum state table size. The reason is that, as explained in Section 5.6.1 and Section D.1, the state machine compares a candidate entry to existing entries. This design has the advantage that it operates on "existing data structures", or rather existing memories, while the latency is acceptable for small maximum allocation table sizes. However, as indicated above, state allocation tables that support 1000 or more entries, would drastically inflate that latency. First, traversing the allocation table to test if an allocation candidate is valid takes much longer. Additionally, a large number means that there probably is a large number of actual states in the system, which increases the likelihood of a collision when testing an allocation candidate. A collision causes the state machine to discard the candidate, determine a new candidate, and traverse the allocation table again, testing the new candidate. Thus, the average latency might even scale close to quadratically in the total allocation table size, instead of just linearly.

The proposed solution for large allocation tables is a "memory-centric" implementation, instead of a "state-centric" implementation. In that case, the memory allocator maintains a new data structure, which represents the occupation status of the physical memory (at a granularity of at least 1-qubit quantum states, or $256\,\mathrm{bit}$) - called the *memory occupation table*, in the following. Regardless of the request and current state space, that new table only needs to be traversed once, in order to find a new allocation.

The proposed scheme lends itself more to many small states, than to few large states. With small states, it is more likely that a suitable "gap" in the memory occupation exists at a relatively low memory address - due to constant allocation and de-allocation - in which case the state machine can terminate the search relatively quickly. A potential downside of large states in memory is that per clock cycle, the state machine can only check a certain number of memory occupation table entries, and thus only a certain memory region size. Therefore, an unnecessary number of cycles would be spent to just traverse the memory region occupied by one state (which in the state-centric scheme is one check). One approach to mitigate this time loss is a hierarchical memory occupation table, that operates at multiple granularities. For instance, for the first $8\,\mathrm{kbit}$, or 32 1-qubit state entries, the table can contain the following bits:

- 32 bits for the 32 individual $256\,\mathrm{bit}$ regions - denoting whether or not they are occupied
- 2 bits for the memory region status - 1 bit signaling "all-occupied", 1 bit signaling "all-empty".

Assuming that timing-wise, the memory allocator can analyze up to 32 bits in one clock cycle (arbitrary number, but potentially realistic because "analyze" here means "find-first-set"), the operation for large states (that occupy at least 32 individual regions) and states smaller than that looks as follows: For small states, the memory allocator looks at 32 consecutive "all-occupied" bits. Afterwards, it only looks into the individual 32 high-granularity bits for regions that have "all-occupied" unset, and tries to find a fitting free memory region. Alternatively, the allocator could also spend one cycle on checking the "all-empty" bits, and immediately select a free memory region, if it encounters one. The projected advantage is a slightly quicker search termination, however at the cost of increased memory fragmentation. This over time can increase latency for large states, and even diminish the advantage for small states, because fewer regions will be "all-empty". For this reason, analyzing the "all-empty" bits is discouraged for allocating small states.

The procedure works the other way around for large states. The "all-occupied bits are ignored", the "all-empty" bits are analyzed first. In this case, it is not even necessary to test the 32 individual region bits, the "all-empty" information is enough. In exchange, there is some overhead to the allocator for detecting large contiguous unoccupied memory regions. Namely, it must retain the starting address of

an unoccupied region, while it tests in the subsequent clock cycles whether that region is large enough to fit the desired state size.

An additional aspect to note is that the algorithm is significantly simpler to implement, if one accepts that states of more than $8\,\mathrm{kbit}$ are aligned in memory to $8\,\mathrm{kbit}$, and that smaller states are aligned to their state size (provided a granularity factor of 32 between hierarchy levels, as described above). This allows large states to fully rely on the "all-empty" bits, and reduces complexity when checking individual $256\,\mathrm{bit}$ regions for small states. Lastly, it is advisable to ensure that states that are smaller than $4\,\mathrm{kB}$ never cross a $4\,\mathrm{kB}$ address boundary, because the memory interface operates via AXI bursts. AXI has a protocol restriction for bursts to not cross $4\,\mathrm{kB}$ address boundaries, such that guaranteeing properly aligned state allocations is beneficial for the memory interface operation, and prevents unnecessary additional bursts (and thus overhead) due to misaligned memory accesses.

7.2.3. Maximum Bandwidth HBM Operation

This section builds on the discussion earlier in this chapter on using a *full-width HBM interface*, in Section 7.1.2. As indicated there, a maximum-throughput design needs to not only utilize all available HBM channels, but also deactivate the HBM's integrated crossbar. Nonetheless, the concept of forming two virtual PC groups from Section 7.1.2, with adjacent pairs of PCs belonging to different groups, carries over to memory operation without the crossbar.

In addition to deactivating the crossbar, maximum throughput entails that **all** available HBM channels are used. This includes the second HBM stack, which in the prototype is dedicated to data exchange with the host via shared memory. In order to free up the second stack for quantum state memory, state data exchange needs to be migrated to shared host memory (which the U55C supports, but older cards do not). A theoretical alternative is enabling host DMA to the entire HBM, and then doing the state representation conversion in software. However, two problems are associated with this approach. First, it is uncertain whether that is possible, with the crossbar being deactivated. Second, it would require the host to directly interact with the accelerator's memory management modules, which is not supported in the current architecture. Therefore, it is advisable to discard the second method, and only attempt maximum-throughput operation on FPGA cards that support shared host memory.

The implication of deactivating the global addressing crossbar is that the AXI channels can not be read-only or write-only any longer, because every PC can only be accessed by one AXI channel. Instead, whether they are read-only, write-only, or need to do both, is decided per operation, depending on source and target memory locations. Section 7.1.2 presented methods to prevent PCs from having to act as both source and target, during an operation, by prioritizing updated allocations in the other respective group, and balancing memory occupation levels between the groups.

It needs to be noted that deactivating the crossbar causes some extra fabric logic resource usage, because a small portion of the crossbar functionality is still required. Namely, engine data bus lanes still need to be able to connect to two pairs of AXI channels. To give an example, with a $256\,\mathrm{bit}$ memory interface, engine data bus lanes 0 and 1 (both send and return datapath) need to be able to connect to PCs 0/1, and to PCs 2/3. For the send datapath, that means an $83\,\mathrm{bit}$ (one state vector element) 2-to-1 multiplexer per lane. For the return datapath, it simply is a fanout 2 data path, because every AXI channel has only one data lane that can potentially access it. This is considered a manageable overhead. The crossbar on the other hand saves this extra hardware, but has a minimum latency penalty of 20 clock cycles [37], even with perfectly local AXI-to-PC accesses (while the potential impact on achievable memory bandwidth is undocumented). The additional hardware resource usage is preferred over the higher latency.

The implementation is not particularly problematic, as long as the extra hardware utilization is considered. Next to the multiplexing, the AXI masters need to be extended. In the prototype, these are hard-coded to either reading or writing, and therefore do not contain the logic to operate in the other direction. The proposed implementation needs both directions, which increases the resources demand of the AXI infrastructure.

7.2.4. Hybrid Quantum State Memory

In the prototype, purely DDR memory is employed for storing quantum states, while most of the FPGA's local on-chip memory resources are left unused. On the one hand, the exponential scaling of quantum state vector size always necessitates a DDR beyond a certain state size, because it can fit significantly larger states than local memories. On the other hand, the same scaling behavior dictates that the vector

size decreases rapidly for small states, such that they fit into typical local memory blocks. Section 6.3.2 has already touched on quantum states in SRAM-like memories, in conjunction with CPU cache. The main advantage over DDR memory is a significantly lower access latency, because no memory page needs to be loaded. The proposal consists in using FPGA local on-chip memory resources to create a "hybrid quantum state memory". In a simple approach, states above a certain qubit size are stored in HBM, states below that size are stored in UltraRAM. The following paragraphs quantify the advantages for small states, and afterwards discuss the choice between different local memories on the FPGA.

Considering access times, AXI-connected HBM has an absolute minimum latency of 108 AXI clock cycles, for closed-page accesses - which is the more realistic case, compared to open page, when reading or writing a state for an operation. BRAM has a deterministic latency of 1 or 2 cycles, depending on output registers, UltraRAM has a non-deterministic latency in the same range. Even under the assumption of a BRAM or UltraRAM block that can only supply one state vector element per clock cycle (in terms of interface bitwidth), a state of up to 6 qubits, or 64 elements, would have been fully read from this memory, before HBM had been able to provide the first dataword. It is realistic to assume that the same holds for 7-qubit states. Beyond this figure, it is expected that the benefits diminish, compared to a full-width HBM interface. Therefore, 7 qubits is considered a good starting point for a hardware validation, in terms of state sizes that are faster to access in local memory than in HBM.

The U55C FPGA offers the choice between BRAM and UltraRAM. The proposal is to exclusively use UltraRAM for hybrid quantum state memory. The main reason lies in keeping BRAM available to tasks that rely on deterministic access latencies and guaranteed bandwidth, such as the improved dense gate array discussed in Section 7.2.5. Additionally, the chip has more UltraRAM capacity than BRAM capacity.

The proposed scheme necessitates several additions to the accelerator's memory back end. The memory allocator, regardless of state-centric or memory-centric allocation (see Section 7.2.2), needs extra logic to differentiate between HBM occupation and UltraRAM occupation. The memory allocation table itself is not affected, provided that the scheme is implemented with a static crossover state size between HBM and local on-chip memory. Eventually, it might however be desirable to incorporate a fallback mechanism for small states to be stored in HBM, if the available space in UltraRAM is used up. The data back end (see Figure 4.7) needs an extension to interact with UltraRAM memory, and to multiplex the engine's source and target datapaths. In summary, while the method does add a certain level of complexity to the memory back end, the necessary logic is manageable, and the expected overall extra resource consumption is small (except for the UltraRAM itself).

Lastly, it needs to be pointed out that hybrid quantum state memory does not offer a noticeable improvement in the current NetSquid front end prototype implementation. Large states are unaffected, and the execution time for small states mostly consists of host-accelerator communication. However, analog to the discussion in Section 7.2.2, future designs that eliminate most of the host-accelerator communication (Section 7.3.3) can benefit significantly.

7.2.5. BRAM-assisted Pairwise Dense Gate Array

This section addresses the severe performance disadvantage of the dense gate array (Section 5.5.3), compared to the software baseline, for states larger than 9 qubits. As explained in Section 6.3.3, the structural cause is the quadratic scaling behavior of the array-based computation. The approach designed in this section does not solve the structural disadvantage, but manages to shift the "crossover" point between software and hardware to significantly higher state sizes, by employing local BRAM buffers. This section is complemented by Section 9.1, which presents an entirely different approach, trading unideal parallelizability and memory access behavior for linear scaling.

The main idea in this section lies in using BRAM blocks as accumulation buffers, which allows each array level to compute up to 512 output state vector element per input vector iteration (instead of 1), and thereby reduces the amount of cycles where a DSP does not actually perform a multiplication. Since it is still an array with a restricted maximum number of output elements per input vector iteration, this design does **not** change the general quadratic scaling behavior for large states. It increases the state size that can be handled in one single read pass, which is what shifts the transition point between linear scaling (with one pass) and quadratic scaling (multiple passes) to larger states. The approach is explained in the following paragraphs, first from an architectural (high-level) perspective, followed by an extensive in-detail discussion per CU, ready to be implemented.

Architecture

Architecturally, the in-DSP accumulation is switched off, and each array level is equipped with its own BRAM accumulation buffer instead. Technically, two physical BRAMs are necessary per array level, one for the real number portion, and one for the imaginary number portion. The remainder of this section will collectively refer to these as one buffer, since they operate in exactly the same way. It is still the *label analyzer* (see Figure 5.6) that is responsible for feeding the correct base gate operation matrix entry to the DSP, upon encountering an input vector label whose amplitude contributes to one of the level's current output vector elements. When a partial result has been computed by the respective DSP, it is written to its associated position in the buffer, or, if necessary, added to an already existing first partial result at that buffer address.

CU Implementation - Output Label Assignment

The central challenge in designing a viable buffer-assisted dense gate array lies in input label filtering. The label analyzer, in a given array level, needs to be able to identify any label that corresponds to the output labels currently assigned to this level's buffer, at a throughput of one input label per clock cycle. At a buffer size of 512 amplitudes (explained below), it is infeasible to individually determine all corresponding input labels, and compare them to a given input label within one cycle. Since the required input labels directly depend on the output labels that are assigned to a buffer, an efficient scheme is required for the output labels, that makes input label filtering feasible.

In the ideal case, the output labels are arranged such that input filtering only takes one bitwise label comparison. As an example, assuming that an input needs to filter $8\,\mathrm{bit}$ numbers for labels $24 - 27$, it is sufficient to check if the 6 MSBs are 000110 (or in 4-state logic language, compare the input label against 000110xx). Formulated as a criterion: Determining whether an input label appears in a set of labels only requires one comparison, if the set of labels is a consecutive range of numbers, aligned to a multiple of a power of 2 $(n \cdot 2^r)$, of a length 2^s with $s \leq r$.

The following sections develop the final output label assignment scheme, by starting with the simplest possible approach, and iteratively increasing complexity, until a feasible scheme is found.

CU Implementation - Consecutive Output Labels

The above criterion motivates a trivial solution, which as the following paragraphs will show has a significant flaw, but is a good starting point. One characteristic of (extended) dense gate matrices is that the main diagonal is always fully populated. Therefore, for a given output element label k^*, it is guaranteed that the input label $k = k^*$ is one of the two input labels that contribute to k^*. With the above observation on consecutive ranges, and assuming buffers to have a power of 2 depth, a straightforward approach would be to simply assign consecutive label ranges to the buffers.

An example illustrates the shortcoming of consecutive output labels. A buffer depth M of 4 is assumed, a 4-qubit state, and a target qubit index $r = 2$ (0-indexed). In this scenario, output labels 0-3 are assigned to the buffer. The resulting pairs for pairwise gate operation (which collectively form the required input labels) are $\{0, 4\}$, $\{1, 5\}$, $\{2, 6\}$, and $\{3, 7\}$. This results in an input element range of 0-7, which matches the above criterion, and is perfectly feasible. A problem occurs for $r = 1$: The new label pairs are $\{0, 2\}$, $\{1, 3\}$, $\{2, 0\}$, and $\{3, 0\}$, resulting in input label range 0-3. Filtering-wise, that is unproblematic. Upon encountering label 0, for instance, there is a resource problem. The amplitude for label 0 needs to be multiplied with two different base gate matrix entries, for the output amplitudes labeled 0 and 2. Since the throughput of one element per cycle can not be compromised, this requires employing two DSPs, instead of one. While technically possible, in the first example ($r = 0$), there is never a task to perform for the second DSP. Therefore, depending on a gate's target qubit, half of the computation array's DSPs would frequently be wasted (idle) hardware.

Whether the associated input label $k \neq k^*$ falls into the same buffer as k^* (provided consecutive output labels) is determined by evaluating $2^r > M$, because $|k_1 - k_2| = 2^r$ for two associated input labels.

CU Implementation - Dual-Stage Array Levels

The problem of "colliding" label pairs can be solved by grouping pairs of adjacent array levels and introducing "stages". From here on, one array level consists of two stages of cascaded DSPs (each with its own BRAM buffer), instead of just one DSP layer (thus double the previous size). The output buffer labels are arranged such that in all cases, corresponding labels fall into different stages of the same

array level. Firstly, that solves the “collision” problem. As a side effect, it saves hardware on filtering. Both stages in one level require the same input amplitudes, albeit with different factors from the base gate matrix (for illustration, if output label 1 requires input labels 1 and 3, then so does output label 3). Therefore, only the first stage in a level needs to implement the filtering circuit, and passes on the label hit information to the second stage (which still is offset by one cycle to the first stage, due to cascading).

In the above example, the adaptation looks as follows: For $r = 2$, labels 0-3 are assigned to stage 0, and labels 4-7 are assigned to stage 1. For $r = 1$, in the above scheme only labels 0-1 would be assigned to stage 0, and labels 2-3 to stage 1. In order to still fully utilize the buffers, the subsequent “batch” is also assigned to this array level, such that stage 0 handles output labels $\{0, 1, 4, 5\}$, and stage 1 handles output labels $\{2, 3, 6, 7\}$. An important side effect of this scheme is that in every label pair, stage 0 holds the smaller label, and stage 1 holds the large label. This characteristic is exploited in the following paragraph.

CU Implementation - Operation Formulas

The formulated scheme makes efficient use of the available buffer space and does not compromise throughput. This paragraph provides formulas for the different components, thereby demonstrating that the proposed scheme is efficient to implement in binary hardware, and that it allows for efficient input label detection.

In the following, the formulas are presented that were derived for implementing the above scheme. The variable nomenclature is as follows:

- M - buffer depth (number of elements in one local BRAM buffer)
- b - position of an element in the buffer (local “address”)
- s - stage (0 or 1)
- l - level (consisting of two stages each)
- r - target qubit index (0-indexed)
- i - input vector iteration
- d - array depth (number of levels in the array)

The required formulas are:

- $k_{\text{out}}(b, l, s, M, r, i, d)$ - the output element label based on its address in the local buffer (beneficial to formulate, because then the label does not have to be stored with the partial amplitude in the buffer, which saves buffer capacity)
- $b(k^*, l, s, M, r)$ - the local buffer address based on the output element label (practically the inverse of k_{out})
- $k^*(k, r, s)$ - the output label corresponding to input label k
- ***factor***(k, r, s) - the base gate operation matrix element to forward to the DSP, upon encountering label k
- ***filter***(l, M, r, i) - the range of input elements to filter for, in a given array level l

Deriving the formulas is aided by Table 7.1 and Table 7.2, which display buffer output label assignment for certain buffer sizes and array levels. It is important to note that several of the following formulas, while consistent with the data from the tables, are yet to be formally proven, or tested on a larger scale.

- $k_{\text{out}}(b, l, s, M, r, i, d)$

$$k_{\text{out}}(b, l, s, M, r, i, d) = 2^{r+1} \cdot \left\lfloor \frac{b}{2^r} \right\rfloor + (b \bmod 2^r) + s \cdot 2^r + f(M, l, r) + i \cdot (2M \cdot d) \tag{7.1}$$

$$\text{with} \quad f(M, l, r) \coloneqq \left\lfloor \frac{l}{g(r, M)} \right\rfloor \cdot g(r, M) \cdot 2M + (l \bmod g(r, M)) \cdot M$$

$$\text{with} \quad g(r, M) \coloneqq \begin{cases} \left\lfloor \frac{2^r}{M} \right\rfloor & \text{if } \left\lfloor \frac{2^r}{M} \right\rfloor \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

$k_{\text{out}}(b, l, s, M, r, i, d)$ describes the data in Table 7.1, and is derived as follows:

l (level)	s (stage)	r (target)	b 0	1	2	3
			output label			
0	0	0	0	2	4	6
0	1	0	1	3	5	7
1	0	0	8	10	12	14
1	1	0	9	11	13	15
0	0	1	0	1	4	5
0	1	1	2	3	6	7
1	0	1	8	9	12	13
1	1	1	10	11	14	15
0	0	2	0	1	2	3
0	1	2	4	5	6	7
1	0	2	8	9	10	11
1	1	2	12	13	14	15
0	0	3	0	1	2	3
0	1	3	8	9	10	11
1	0	3	4	5	6	7
1	1	3	12	13	14	15

Table 7.1: Output label assignment for buffer positions b, at buffer depth $M = 4$. Supplementary table for deriving implementation formula.

M	r	l 0	1	2	3	4	5
		buffer label for $b = 0$					
2	0	0	4	8	12	16	
2	1	0	4	8	12	16	
2	2	0	2	8	10	16	18
2	3	0	2	4	6	16	
4	0	0	8	16	24		
4	1	0	8	16	24		
4	2	0	8	16	24		
4	3	0	4	16	20		
8	0	0	16	32	48		
8	1	0	16	32	48		
8	2	0	16	32	48		
8	3	0	16	32	48		
8	4	0	8	32	40		

Table 7.2: First stage ($s = 0$) first buffer element ($b = 0$) output label assignment, at various buffer depths M. Supplementary table for deriving implementation formula.

  - $2^{r+1} \cdot \lfloor \frac{b}{2^r} \rfloor + b \bmod 2^r$ is the term for $l = 0$ and $s = 0$.
    - $2^{r+1} \cdot \lfloor \frac{b}{2^r} \rfloor$ is a "binning type" operation, that yields the smallest possible number of buffer positions b, such that the two stages of one level collectively hold a consecutive range of output labels (compare Table 7.1).
    - $b \bmod 2^r$ is the offset of a given label within said group.
  - $s \cdot 2^r$ refers to the distance between two related indices always being 2^r, while stage 0 always holds the smaller label in a pair.
  - $f(M, l, r)$ is the first label ($b = 0$) in a buffer, and therefore describes the data in Table 7.2.
    - $g(r, M)$ is the number of consecutive levels l who have "interleaving" between their output labels (before a new consecutive range of output labels begins). For instance, in Table 7.1, at $r = 3$, levels 0 and 1 (l) collectively hold a consecutive range of output labels ($0 - 15$), but $0 - 3$ is in level 0, $4 - 7$ in level 1, $8 - 11$ again in level 0 etc., hence "interleaved". Afterwards, levels 2 and 3 (not displayed) have interleaving output labels, but collectively form the consecutive output label range $16 - 31$. Accordingly, $g(r = 3, M = 4) = 2$. For $r \leq 2$, in Table 7.1, $g = 1$, thus no interleaving between levels.
    - $\left\lfloor \frac{l}{g(r,M)} \right\rfloor \cdot g(r, M)$ acts like a "binning" operation, the term groups labels according to $g(r, M)$, and yields the group's first "level index" for all labels in that group. A level has $2 \cdot M$ buffered elements (because of two stages), hence the additional factor $2M$ in $f(M, l, r)$, to obtain the "global" first output label in a group.
    - $(l \bmod g(r)) \cdot M$ yields the offset of the first element in a level, within its "group" (note that only one buffer depth M counts, because the offset of the first level's stage 1 buffer is larger that of the last level's stage 0 buffer, in a group).
  - $i \cdot (2M \cdot d)$ adds the global label offset for the iterations of the input vector, because the array in total can buffer $2M \cdot d$ amplitudes.

- $b(k^*, l, s, M, r)$

$$b(k^*, l, s, M, r) = h(k^* - s \cdot 2^r - f(M, l, r) - i \cdot (2M \cdot d)) \tag{7.2}$$
$$\text{with} \quad h(x) \coloneqq 2^r \cdot \left\lfloor \frac{x}{2^{r+1}} \right\rfloor + (x \bmod 2^r)$$

- $h(x)$ is effectively the inverse to the term $2^{r+1} \cdot \lfloor \frac{b}{2^r} \rfloor + b \bmod 2^r$ in $k_{\text{out}}(...)$. Another perspective is that it describes exactly the case $s = l = 0$.
- The inner term "reduces" every element to the corresponding buffer position $s = l = 0$ (compare $k_{\text{out}}(...)$).

- $k^*(k, r, s)$

$$s = \begin{cases} 0 \ : \ k^* = k \,\&\, \sim (1 << r) \\ 0 \ : \ k^* = k \,|\, (1 << r) \end{cases} \tag{7.3}$$
$$\text{with} \quad \& \ : \ \text{logical AND}, \quad | \ : \ \text{logical OR}, \quad << : \ \text{logical left-shift}$$

- *factor*(k, r, s)

$$\text{in the base gate matrix:} \quad \text{row} = k[r] \ , \quad \text{column} = s \tag{7.4}$$

- *filter*(l, M, r, i)

filter range: (ranges in Verilog notation)

$$\begin{cases} [((i \cdot 2M \cdot d) + f(M, l, r)) + : 2M] & \text{if } 2^r \leq M \\ [((i \cdot 2M \cdot d) + f(M, l, r)) + : M] \cup [((i \cdot 2M \cdot d) + f(M, l, r) + 2^r) + : M] & \text{if } 2^r > M \end{cases} \tag{7.5}$$

- The ranges adhere to the criterion formulated above:
  - $f(M, l, r)$ is a multiple of M, which in turn is a power of 2.
  - For $2^r \leq M$, $g(r, M) = 1$, therefore $(l \bmod g(r, M) = 0$, and $f(M, l, r)$ is even a multiple of $2M$.
  - Therefore, $((i \cdot 2M \cdot d) + f(M, l, r))$ is always a multiple of M, and a multiple of $2M$ if $2^r \leq M$.
  - With M a power of 2, $2^r > M$ implies that 2^r is a multiple of M.
  - Consequentially, $((i \cdot 2M \cdot d) + f(M, l, r) + 2^r)$ is a multiple of M, for $2^r > M$
  - Due to the given argumentation, both cases in Equation (7.5) match the criterion on efficient binary filtering for consecutive number ranges. The filtering circuit therefore needs at maximum two parallel bitwise comparisons, combined by a logic OR, to determine whether or not an input label is a hit for the level's current output label range(s).

While these formulas appear relatively complex and unpractical in algebraic notation, their corresponding expressions in binary hardware are comparably simple - as long as M and d are a power of 2. That is because the formulas are almost entirely composed of the expressions showcased in Figure 7.1, which as shown in that overview are significantly more complex algebraically, than they are in bitwise expressions. Additionally, chaining bitwise operations makes it easy for the synthesis tool to internally optimize and reduce long formulas. The multiplication by i is unproblematic, since it only denotes different iterations. In practice, it is therefore not a multiplication, but rather an addition to a constant bias, once per iteration.

It is important to note that these formulas are practically independent from each other, which helps with hardware implementation. *Filter* is a "hit or miss" function. It is not important which exact label was detected - as long as the input label k is forwarded (via fabric logic) to the DSP output side, alongside with the information label hit information (which only needs to be done per level, not per stage). In contrast to the existing non-BRAM array design, in the optimized design there is no need to feed a zero to the DSP input in case of a miss. The decision whether or not to ignore an input amplitude is taken after the DSP, based on the propagated *filter* hit information. That makes the *factor* function completely parallel to the *filter* function - splitting up and simplifying one of the prototype's critical timing paths (see Section 6.1). Post-DSP, theoretically, $b(...)$ depends on $k^*(...)$, but since $k^*(...)$ is trivial, and effectively hard-coded, that is not a problem in practice.

algebraic expression	verilog code	
2^s	`1<<s`	only set bit s (0-indexed)
$x \cdot 2^s$	`x<<s`	logical left-shift x by s bit positions
$x \bmod 2^s$	`x & {s'{1'b1}}`	take only the s least significant bits of x (verilog expression `{s'{1'b1}}` does not synthesize because s is not static, but `(1<<s)-1` is equivalent and synthesizes)
$\left\lfloor \frac{x}{2^s} \right\rfloor$	`x>>s`	logical right-shift on x by s bit positions
$2^s \cdot \left\lfloor \frac{x}{2^s} \right\rfloor$	`(x>>s)<<s`	set the s least significant bits of x to 0 (`x & {($bits(x)-s)'{1'b1},s'{1'b0}}`)

Figure 7.1: Algebraic building blocks in implementation formulas, and their bitwise Verilog expressions

CU Implementation - Hardware Layout

The proposed scheme lends itself well to the hardware layout on (Xilinx) FPGAs. The DSP48E2 and BRAM18B DSP and BRAM primitives are designed and arranged to be combined with each other. They have the same height in the chip, and form parallel columns, at close physical proximity. Therefore, one can assign exactly one ($18\,\mathrm{kbit}$) BRAM18B unit to one DSP48E2 unit. It is imperative here to use BRAM, and not UltraRAM, for instance, because BRAM has fully deterministic latency - indispensable in a fixed-throughput application - and supports dual-port operation - which also is crucial, as will be explained below. The maximum port width of a BRAM18B unit, and thus the maximum state vector element amplitude width, is $36\,\mathrm{bit}$. In order to offer this bitwidth, the memory needs to operate in Simple Dual-Port (SDP) mode - meaning equal bitwidth for read and write port, as opposed to True Dual-Port (TDP) - which is consistent with the application here. With a $36\,\mathrm{bit}$ interface, it is logical that an $18\,\mathrm{kbit}$ memory has 512 entries, thus $M = 512 = 2^9$. The maximum interface width of $36\,\mathrm{bit}$ reinforces the need to formulate $k_{\mathrm{out}}(...)$, because $36\,\mathrm{bit}$ is enough to store a $27\,\mathrm{bit}$ amplitude, but not enough for an additional label.

In theory, $M = 1024$ is expected to be feasible, by employing two BRAM18B per stage. That is because per stage (and array lane), two DSPs compute partial results for the real part of the result amplitude (the products of the two real parts, and of the two imaginary parts, of input amplitude and base gate matrix entry), and for the imaginary part. Since these originate from the same label, the partial results can be accumulated before (or while) writing to the BRAM buffer. This way, only one physical BRAM serves two DSP columns. The (otherwise unused) second column's buffer can be used to extend the first column's buffer, which doubles M. Nevertheless, calculations in this section are generally based on the conservative assumption of $M = 512$. For $M = 1024$, one qubit needs to be added to any result, in terms of state size.

CU Implementation - Continuous Dataflow

There are two challenges to the design, in conjunction with serving a continuous dataflow, including several input vector iterations, or subtransmissions.

The first potential problem lies in read and write congestion at the buffers, and is solved by the BRAM's ability for dual-port operation. When a partial result is computed by the DSP, and its address b in the buffer has been determined, the following steps need to happen:

1. Load the current entry at b, because it is possible that there already is a partial result for that output label.
2. If applicable, add the new partial result to the existing one.
3. If there already was a partial result: Write the accumulation result to the output data stream, write an "invalidated" entry buffer position b, for the next iteration.

   If there was no partial result: Write the partial result to buffer position b.

Evidently, since all these steps need to happen at a throughput of one element per cycle (not necessarily within one cycle), the buffer must support exactly one read and one write operation per cycle.

The second challenge lies in "cleaning out" the buffer in time, before the second iteration. Although the answer has already been given in the previous paragraph, this paragraph briefly clarifies the potential problem: In the existing dense gate array, it is sufficient to write out the levels' results at the end of every iteration, in order to free up the accumulation buffer (the DSP output register, in that case) for the next iteration. With a buffer size larger than 1, that does not work, because subsequent cycles can already belong to different iterations, such that any buffer must be fully empty at the start of every iteration. The problem is solved by the technique indicated above. One bit in the buffer entries is designated as an `empty` bit, which is unset upon storing the first partial result, and set after adding the second partial result. This way, the circuit can distinguish between the first and second partial result, within an iteration, and immediately write a result to the output once the second partial result is available. As a result, the buffer is guaranteed to be empty after the last cycle of an iteration.

Speedup Projection

With a full 16-channel, $256\,\mathrm{bit}$-scaled memory interface, quadratic scaling begins at 6-qubit states with the prototype architecture, and at 15-qubit states with this section's proposal. In other words, the "single-pass capacity" of the BRAM-assisted array, scaled to 16 channels, is 15 qubits (or 16 qubits, with $M = 1042$). In Figure 6.6, the performance crossover point between accelerator and software baseline is at 9 qubits, which is 7 qubits above the (single-channel prototype) array's single-pass capacity. Additionally, that figure is obtained with the throughput of a 1-channel congestion-affected memory interface, instead of a 16-channel memory connection designed to avoid congestions. The throughput increase factor of 16 translates to 4 extra qubits, in the same time. The extrapolation leads to an expected runtime crossover point at a state size of $15 + 7 + 4 = 26$ qubits (27 qubits with $M = 1024$).

7.3. Schematic Functionality Extension

7.3.1. Data-Compressed Labeled Vectors

This section formulates a proposal to combine the advantages of this work's accelerator design with those of a related labeled state vector representation. Referring back to Section 2.4.2, labeled-hashed vector computation, by Jaques et al. [18], also utilizes labeled state vector elements, and the same computation idea (pairwise gate operation and element update formulas) as this work. The main topic of their publication however lies in "leveraging state sparsity", which in their case means treating the state vector as a sparse vector. In doing so, they gain the ability to perform non-noisy simulations on quantum states of 100 and more qubits, which would otherwise by far exceed any available memory capacity. This idea can be applied to the presented accelerator design as an alternative state vector representation, which significantly enhances the number of simulation scenarios that can benefit from the accelerator.

Instead of using the labeled-hashed representation that Jaques et al. introduced, the proposal is to derive a *labeled-reduced* state vector representation. The difference is that labeled-reduced state vectors would only contain non-zero amplitudes, like labeled-hashed vectors, but be stored in random order, in contiguous memory, like labeled-random vectors. Another way of describing it is removing any non-zero amplitudes from a labeled-random vector, and "compressing the remaining vector to fill the gaps". The advantage of labeled-reduced vectors over labeled-random vectors is that they can operate with fully sequential memory accessing - like labeled-random vectors - and that occupying contiguous memory regions facilitates managing multiple states in memory. The original publication on labeled-hashed vectors [18] only concerned Quantum Computing simulations, not Quantum Network simulations, and therefore had no need to support multiple states in one memory.

The integration of labeled-reduced vectors introduces complexity to the on-chip memory management, but is relatively unproblematic computation-wise. For illustration, from the engine's perspective, there is almost no difference between a 4-element labeled-reduced vector, and the first 4 elements of a 16-element labeled-random vector. Nearly every engine module simply processes input elements one-by-one, from the engine data bus `transmission start` signal, until the `transmission end` signal. Concerning memory management however, the "state size" (number of qubits) does not automatically determine the memory space occupied by a state, which on top of that is also no longer guaranteed to be a power of 2.

For the engine, adaptations are required in the tensor and dense gate modules. The tensor module needs a way to identify the last transmission of the streamed state 1, in order to correctly terminate

the output engine data bus transmission (see Section 5.5.2). Currently, it relies on calculating the number of state 1 and state 2 transmission, by means of their state sizes (in numbers of qubits). In a labeled-reduced scheme, the relation between qubits per state and state vector elements is unknown. One solution to replace the current operation termination mechanism is extending the engine data bus protocol by an `operation end` signal. This can either be a separate control signal, or - the preferred option - be implemented with existing signals, by defining that asserting `end & ~start` for 2 cycles, instead of 1 cycle, does not only end a transmission, but also an operation.

The dense gate module is another problem, if realized with local BRAM buffers as proposed in Section 7.2.5. It does not know anymore when to write a buffered element to the output data bus, which causes the buffers to not empty in time for the next input vector iteration. With labeled-random vectors, buffered output elements are written to the output data bus, as soon as the second partial result has been added, which ensures that the buffer is completely empty after the last cycle of one input vector iteration - and thus ready for the next iteration. In the labeled-reduced scheme, for a buffered partial result, there is no way to tell for the module if there will be a second partial result. In the event that an input label k_1 was processed and buffered (in both stages), but the corresponding input label k_2 is a zero-element and thus never appears, the buffered k_1^* and k_2^* remain in the buffer until the end of the transmission, although they are final results, and block the buffer for the next iteration. The proposed solution - although not ideal - is to suspend the input vector reading in between of iterations, for just enough cycles for the BRAM buffers to be emptied. This requires another extension to the engine data bus protocol. In order to not introduce new control signals, the rule can be added that setting `start & ~end` during a transmission suspends the (sub)transmission. A subsequent `start` resumes the transmission, subsequent `start & end` or `end` act normally, terminating a transmission or subtransmission. With maximum-depth buffers, the solution introduces 512 cycles of latency between transmissions, which at $450\,\mathrm{MHz}$ equates to about $1.138\,\mu\mathrm{s}$.

In terms of memory management, both the memory allocation table and the memory allocator are affected. The allocation table requires an additional field, which holds the actual state size in memory. Allocating memory can not be done any longer by number of qubits, but needs to be done by number of elements in the state (denoted as n in the following). Additionally, n is likely to not be a power of 2, in many, and also allocations are not necessarily aligned to powers of 2. Both complicates finding and testing a candidate allocation, because in a binary number system, arithmetic operations on numbers that are guaranteed to be powers of 2 are significantly simpler.

It remains for future exploration to find the most adequate solutions to these challenges. This paragraph names several aspects that appear to be promising starting points, for a complete future scheme. First, allocating operation target memory for quantum state operations can be simplified by the fact that for every operation, there is a certain degree of connection between input and output number of non-zero elements. Monomial gates have exactly one output element per input element, so n is constant. Tensor operations have $n^* = n_1 \cdot n_2$ output elements. Dense gates have $n \leq n^* \leq 2n$ output elements (because it is unknown how many pairwise operation label pairs are among the non-zero elements). Measurements have at most n output elements. Therefore, there is a reasonably good upper boundary for every operation type, which enables finding an allocation that might be too large by some amount, but is guaranteed to be large enough. Additionally, it is expected that restricting allocations to “hardware-friendly” granularities has more benefits by reducing complexity, than downsides due to unused memory space. The number of elements in an allocation should always be at least a multiple of a certain power of 2, if not a power of 2 itself.

Concluding, the memory space efficiency of labeled-hashed state vectors can be combined with the accelerator’s computation engine almost “as is”, with not trivial, but most likely feasible adaptations to the on-chip memory back end. This opens up the existing accelerator design to simulations on non-noisy quantum states, that are larger than what the available memory would otherwise allow. The exact maximum state size depends on the state vector sparsity.

7.3.2. Operation Parallelism

Since the engine is designed as a system of multiplexed circuits, which are each responsible for one specific type of operation, a majority of the modules is always idle, because the accelerator can only handle one operation at a time. There are however quantum simulation scenarios which theoretically allow for multiple operations to be performed at the same time, and thus keep a larger portion of the engine active. This section is dedicated to exploring quantum state operations executed in parallel. It

needs to be stated however that, as the following paragraphs will show, the benefits here are likely to not outweigh the cost, in the form of additional implementation complexity and chip resource usage.

In terms of simulation scenarios, the idea practically only has relevance for Quantum Networking, not for Quantum Computing. That is because for operations to be executed at the same time, they need to be fully independent from each other, which implies that there can be no overlap between the states involved. Quantum Computing simulation however typically only entails one entangled state. In Quantum Networking, it is not uncommon to have various nodes perform operations on disjoint states, at roughly the same simulation time. Examples are a long repeater chain, before the end-to-end entanglement is established, or a network simulation with a high number of endpoint nodes.

From an implementation point of view, architectural extensions are required to the host-accelerator interface, and to the on-chip memory interface. The host-accelerator interface requires a mechanism to handle operations as operation requests, either in software, or in hardware. It needs to determine whether an incoming operation request interferes with an ongoing operation, and only trigger the operation execution if there is no conflict. The memory quantum state memory unit (Figure 4.7) would require more severe changes. Most problematically, the data back end needs to be equipped to serve two round-trip data streams to the engine at once. The best technique to do that is yet to be determined. One approach is separating the main memory into 4 PCs groups, instead of 2, such that two disjoint pairs of PC groups serve one operation each. Nevertheless, the system still needs to be able to route from any PC group to any engine module, resulting in a large fabric logic crossbar structure. Another possibility, provided that hybrid quantum state memory is employed (Section 7.2.4), is to only allow concurrent operations from different memory types. This however requires communication between the scheduler and the memory allocation table. It is also considered impractical in practice, because same-time disjoint operations usually happen on relatively small states, which therefore are likely to reside in the same memory. Secondly, access from the control block FSMs to memory management, and potentially also data back end, would need to be implemented token-based, instead of simple static multiplexing (per operation). That is because one operation might need access to the memory allocation table, and memory allocator, while another operation is in progress. Memory management accesses however need to be atomic operations, from the control block FSMs' perspective.

In conclusion, the previous paragraph indicates no clearly promising path for implementing parallelism at operation level. The main problem is the far-reaching intervention into the memory back end architecture. This does not only impose implementation complexity, but also cause significantly higher hardware usage and worse routability, and potentially interferes with non-parallel execution performance. It is considered unlikely that there is a meaningful number of simulation scenarios that justify this overhead.

7.3.3. On-Chip Instruction Management

As indicated in Section 6.3.4, the prototype implementation experiences substantial overhead from hast-accelerator communication, compared to the pure in-hardware computation time of small-state quantum operations. The discussion in Section 6.3.4 furthermore suggests that there are simulation scenarios which allow eliminating this overhead, thereby enabling significantly higher speedups for operations on small states. This section is dedicated to exploring these kinds of simulations.

Two types of on-chip instruction management are distinguished, with increasing functionality set and complexity. The first section discusses almost purely handling quantum instructions, which lends itself to Quantum Computing simulation scenarios. The second section additionally considers higher-level control and message flow, which is imperative for Quantum Networking.

On-Chip Instruction Queue

In order to free the simulation of the constant communication between accelerator host, the operations to be executed need to be transferred to the accelerator in one batch. The accelerator afterwards handles these as an instruction queue, or as a quantum program, in the same way that a conventional CPU executes a program from instruction memory. This way, the FPGA does not need to receive operations from the host application one-by-one.

The proposal is to implement a "quantum ISA processor" in the FPGA, which interprets a suitable variant of Quantum Assembly Language (QASM) [43]. QASM is an attempt to define a unified, assembly-like language for quantum state operations, which has been widely adopted. Numerous variants of QASM have been defined, among which OpenQASM 2 and 3 [44, 45], which among other features facilitate

gate optimization, and NetQASM [46], which is part of the next section. Advantages of employing QASM are that it is a ready-to-use instruction set, which exactly suits the purpose of defining quantum program to execute for the accelerator, and that several existing simulators have built-in QASM interfacing. As an example, IBM's popular simulator Qiskit [47] supports exporting to OpenQASM 3.

The required steps are

- identify the most suitable QASM variant
- define a binary instruction encoding
- implement the processor (that in turn controls the existing accelerator architecture)
- establish a method for the accelerator to receive a quantum program from the host application, and to transmit back results

In terms of exchanging a quantum program, it is suggested to use shared host memory, from which the kernel fetches into a local on-chip instruction buffer. Ideally, of course, all instructions would be stored locally. Both BRAM and UltraRAM are suitable for the buffer, the decision should be based on resource usage in the rest of the accelerator. If available, BRAM is preferred, because of the guaranteed low access latency, and because available BRAM would otherwise be unused. Using UltraRAM would offer higher density, but always take away memory from small-state hybrid quantum state memory (see Section 7.2.4), and potentially have slightly higher access latency, although most likely to a negligible extent. The intermediary step via shared host memory allows efficient handling of programs that are too large for the local buffer. The chip's available DMA resources for large programs (HBM and UltraRAM) are occupied for quantum state memory. Therefore, large quantum programs need to either use shared host memory (where the latency can be hidden via pre-fetching), or the program needs to be transmitted from host to accelerator in batches, via DMA into the local instruction buffer memory. It is evident that the second method incurs significantly more overhead than using shared host memory, such that shared host memory is the desirable method - provided that the FPGA accelerator card supports shared host memory, as the U55C does.

Implementation-wise, it is advisable to consider the consequences on floorplanning in the FPGA, when adding a unit like the proposed processor. The best physical location in the chip for the processor is almost certainly SLR 2. This is because a full-width memory interface is going to take up a substantial portion of SLR 0, where the memory is located, and the engine modules should be placed as close as possible to the memory interface, to limit the complexity of on-chip data movement - thus in SLRs 0 and 1, if possible. The processor only interacts with the quantum state computation portion via an instruction interface, which means small amounts of data, with no fixed latency or throughput constraints. This kind of data is significantly simpler to pass through the chip, than a wide data bus. Therefore, processor and memory interface should be placed as far as possible from each other, such that the engine can be placed close to the memory interface. The implication is that adding the QASM processor comes at the additional complexity of careful floorplanning, and SLR crossings.

An additional aspect to consider, for a potential implementation, concerns system clocking. As explained in Section 5.1.2 and Section 7.1.1, the intended eventual kernel clock frequency is $450\,\mathrm{MHz}$. Since a "processor-like" module at that frequency is at least challenging, if not unrealistic, it is considered likely that the processor clock needs to be separated from the kernel clock. While not particularly problematic - given there is no data traffic between instruction processor and kernel, only instruction traffic - it still slightly increases implementation complexity.

It comes with some uncertainty when predicting the speedup using an on-chip instruction processor, over a baseline like software-based NetSquid `qubitapi` execution. However, the data in Section 6.3.4 and Section 6.3.6 gives a certain indication. Section 6.3.4 extrapolates no-overhead speedup factors for 3-qubit operations between 25 and 120, depending on the operation type. With the instruction processor, these speedups become a realistic target, instead of a theoretical figure. Additionally, the extrapolation considers nor hybrid quantum state memory (Section 7.2.4), which greatly reduces access times for small states, nor full-width HBM interface extension (Section 7.1.2) or an improved dense gate module. The latter two increase system performance for large states, which are commonly seen in Quantum Computing simulations. For these, Section 6.3.6 can serve as a reference, which extrapolates a potential speedup factor of roughly 50 for simulations on 22-qubit states. Combined, the data suggests that Quantum Computing simulations, at state sizes of up to at least 26 qubits, can execute roughly 50 times faster on the improved accelerator, than in the software baseline.

7.3.4. Quantum Network Node Simulation

Building on the idea of a QASM processor for controlling on-chip Quantum Computing simulations, this section explores the option of a similar design for Quantum Network simulations. The most promising approach, as indicated in the previous section, is using NetQASM [46], as the QASM variant of choice. While that seems to be a convenient solution at first, it adds a considerable amount of complexity to the hardware implementation - partially due to the simulation scenarios, partially due to NetQASM itself - as the following paragraph shows.

The first challenge with on-chip Quantum Network simulation is the multi-agent nature of a network simulation. Typically, multiple simulated quantum network nodes each run a certain program, or protocol, independently from each other. A hardware implementation needs to be able to reflect this, for which it has two options. Either the nodes can be simulated by separate hardware instances, and for instance connect to one central messaging bus, which allows them to communicate with a central quantum state computation back end. The computation unit needs to be central, because nodes share states, without even knowing with which node exactly they share a state. The second method is one multi-thread processing unit, which simulates multiple nodes via context switching. The latter is the preferred approach, for two reasons. First, separate hardware units do not scale to large simulations because of limited chip resources. Secondly, it is likely that separate hardware units would be an inefficient use of logic resources. Executing operations in the computation back end is assumed to take significantly longer than decoding instructions and handling non-operation tasks in a NetQASM processor. Therefore, the separate units would spend most of the time waiting for the computation back end.

The second challenge is the complexity that NetQASM adds to QASM. For instance, nodes in the NetQASM execution model have access to classical memory, for storing multiple measurement results, or even performing small arithmetic operations. The processor therefore requires additional on-chip memory resources, which leads to the same availability problems as discussed for local on-chip instruction cache in the previous section. Moreover, NetQASM supports limited node-to-node communication, for quantum entanglement generation. This necessitates a simplistic message exchange system between threads, with a thread wake-up mechanism. Lastly, simulated nodes are still assigned to applications (in the sense of the NetQASM execution stack) within the host application layer. Accordingly, the host-accelerator interface needs an extension to handle host-accelerator communication for multiple (virtual) threads.

Concluding, an on-chip Quantum Network instruction management via NetQASM is feasible, but creates a substantial amount of complexity during implementation. In return, it promises significantly faster Quantum Network simulation. The expected speedup comes from eliminating a huge portion of the prototype's host-accelerator communication, as well as from faster context switching and elementary messaging between simulated nodes, due to a small, dedicated processor, instead of a generic host CPU.

NetSquid Applicability

Since NetSquid was used as the demonstration front end, this section elaborates on whether a (typical) NetSquid simulation can benefit from the proposed on-chip NetQASM implementation. Unfortunately, it is considered unlikely that NetSquid can significantly benefit from a NetQASM-based accelerator. This is due to a too small portion of accelerated code, while NetSquid's architecture does not allow increasing that portion.

In terms of accelerated code, the only effect of a NetQASM simulator would be reducing communication overhead during the accelerated code portion - provided that the simulation's NetSquid node protocols are written NetQASM-compatible, which is not a given. It would therefore not affect which part of the code can be accelerated, which leaves the system with still only about $30\,\%$ runtime savings at best (for a repeater chain), as described in Section 6.3.6.

In order to achieve a meaningful speedup, the portion of NetSquid to be offloaded to the accelerator would need to be increased significantly. The profiling data in Figure 6.10 (from [13]) illustrates most of NetSquid's runtime is spent simulating *components* (and their *models*), and that another noteworthy contribution comes from the *pydynaa* package, which is responsible for virtual event handling. In theory, the accelerator would require the ability to model NetSquid components - potentially in the same way as it would handle NetQASM, via a context-switching, dedicated processor. In practice, that is a substantial challenge implementation-wise, and virtually incompatible with NetSquid's generic python user interface. The hardware implementation would need to handle things like component communication, noise models,

and NetSquid's simulation time model. It is effectively a re-implementation of the simulator, on FPGA. Considering the user interface, NetSquid purposely allows specifying component behavior and protocols in arbitrary python code. This makes the simulator accessible, and highly flexible. Running arbitrary python code however would explode the complexity of the on-chip NetSquid component processor, and even if feasible would be much too slow for a simulation speedup. Accordingly, it would be unavoidable to compile NetSquid protocols to the (reduced) instruction set of an on-chip component emulator.

In summary, a meaningful NetSquid simulation speedup (for the example of a repeater chain) can only be achieved by migrating NetSquid's entire execution model to the FPGA, which in itself comes with great difficulties. Additionally, it would require a method to compile NetSquid simulation scenarios to a restricted accelerator ISA. It is projected that ISA restrictions would translate into constraints on the description and functionality set of simulations, in order to allow compilation, and that due to the extra initial compilation time, only large simulations would benefit from the acceleration.

7.4. Conclusion

This chapter thoroughly studied potential future development steps of the presented accelerator, both in terms of improving the existing functionalities, and introducing new functionalities. While some additions lean towards specific simulation scenarios, others can be seen as global improvements, which apply to any usecase. As a third category, some seemingly promising extensions were studied that turned out not advisable to pursue.

The first section, Section 7.1, performed an in-detail discussion on improvements that had already been named during the thesis, and are practically part of the existing design. The discussion entailed increasing the system-wide clock frequency to the HBM's maximum throughput frequency of $450\,\mathrm{MHz}$, extending the memory interface to using all 32 HBM channels, and improving the accuracy of the engine's inverse square-root computation, by means of a more detailed parameterization and a doubled DSP multiplier width. All of these were eventually categorized as globally useful, and feasible to implement without substantial problems or downsides. These improvements should therefore have high priority in future project development.

Section 7.2 discussed optimizations that had not been named yet, and concern existing system modules. Categorized as globally useful, and ready to implement, were adapting the memory interface to the physical memory (Section 7.2.1) - which saves hardware without throughput loss - enhancing the dense gate computation array with local BRAM buffers - found to increase its applicable computation range by up to 10 qubits - and employing background memory allocation (Section 7.2.2) - saving memory allocator latency. A supplementary proposal was formulated in Section 7.2.2 for a memory-centric memory allocation mechanism, instead of allocation table-centric. It is expected to reduce latency in any scenario that does not purely consist of large states, or generally is compiled for allowing a large number of states in memory, at low overhead, and is therefore considered advisable to implement. It was furthermore laid out how on-chip UltraRAM can complement HBM as low-latency quantum state memory (compared to HBM), which can reduce the round-trip computation times for small states to below the pure memory read access times for the same states in HBM. The method was found to be effective, at reasonable implementation complexity, but only offers an actual advantage when eliminating the per-operation host-accelerator communication latency (as proposed in Section 7.3.3), in which case it provides significant advantages on state sizes below 7 or 8 qubits. Lastly, Section 7.2.3 discussed the possibility of operating the HBM not only at full interface width, but also without global addressing, for maximum throughput. It was concluded that a preliminary benchmarking experiment needs to be carried out, in order to determine if the speedup over crossbar-enabled full-width HBM operation justifies the additional hardware cost, and the slightly more complex implementation.

The final section, Section 7.3, was dedicated to adaptations that impact the application front end. Most importantly, Section 7.3.3 discussed how an on-chip instruction processor for an established standard like OpenQASM can enable significantly more efficient Quantum Computing simulation and greater accessibility, although at considerable additional design complexity. Categorized as highly usecase-specific options were an on-chip processor for a Quantum Network instruction standard like NetQASM, and the adaptation of labeled-hashed state vectors [18] for very large state simulation. Both have applications, but come at substantial design complexity (and manageable extra hardware usage), such that the necessary implementation effort must be weighed against the number of benefitting simulation scenarios. Lastly, studies into a higher-layer NetSquid integration for relevant small-state

simulation, and into potential on-chip instruction parallelism, led to unpromising results. Instruction parallelism, for a higher engine utilization, hardly justifies the associated design and hardware complexity, and has potential negative effects on maximum (non-parallel) throughput. More efficient NetSquid acceleration would require an essential re-implementation on FPGA, which is considered technically feasible, but introduces severe complexity and a long design phase, while only one simulator would benefit.

In summary, the highest-priority improvements are found to be a BRAM-aided dense gate array - which solves a huge portion of the prototype's defining performance issue - alongside with clock frequency optimization, full-width HBM operation with interface width adaptation, and inverse square-root accuracy optimization. While background memory allocation is also a global improvement, and unproblematic to implement, its quantitative impact is significantly smaller than that of the aforementioned additions. For the subsequent priority level, or development iteration, it is proposed to bundle an OpenQASM interface, memory-centric memory allocation, and hybrid quantum state memory. Left for exploration are maximum-bandwidth HBM operation, and data-compressed labeled vectors.

8

Conclusion

8.1. Summary

Chapter 2 first introduced necessary fundamentals on quantum state vector simulation, and on (DSP-based) accelerated computing on FPGAs. Afterwards, a study was conducted on existing quantum computation simulators on FPGA, finding that most designs exhibit severe scaling drawbacks, and lack flexibility. Additionally, the survey revealed that no FPGA design so far takes steps towards translating software-proven methods for efficient state vector computation to hardware.

Chapter 3 introduced the novel *labeled-random* state vector computation scheme, including hardware-efficient element update formulas to replace gate operation matrices. It was demonstrated how this technique can significantly help to exploit modern, parallelized accelerator platforms for universal gate-level quantum simulation, in a way that related approaches are not able to.

Chapter 4 presented a parallelizable, modular high-level design for a hardware accelerator, built around labeled-random state vector representation and element update formulas. An FPGA was chosen over a GPU or a multi-core CPU as the target platform for this work, due to great scalability in computation parallelization and memory bandwidth, and because of the great flexibility for future design space exploration.

Chapter 5 demonstrated how the high-level accelerator design for labeled-random state vector computation was realized as a prototype system on an AMD Alveo U55C FPGA accelerator card. Every relevant aspect of the implementation was explained in detail, from an efficient implementation-wide fixed-point complex number representation, through implementation complexity and locality optimization via a lightweight computing core design and custom main data bus, back end memory management and modularity, to on-chip hardware component orchestration, and generic front end software interface.

Chapter 6 presented any relevant findings concerning the implemented prototype. The accelerator was found to operate correctly and with good accuracy, while the FPGA implementation data indicates no scaling issue for the prototype in terms of data bandwidth, functionality set, or clock frequency. Performance-wise, it was found that the accelerator outperforms a software HPC CPU baseline by roughly factor 2 to 5 (depending on the operation), with an increase to factor 4 to 15 for state sizes beyond the CPU's cache memory capacity. *Non-monomial gates* (see Glossary) however only outperform the baseline for very small states, in the current implementation, due to the quadratic scaling behavior of the in-hardware computation array, and its small size in the prototype. Under the assumption of a more efficient future dense gate implementation, extrapolating to a full-width HBM memory interface led to a projected average large state-operation speedup factor of 93, or 162 with optimized kernel clock frequency.

Finally, Chapter 7 thoroughly studied potential future development steps of the presented prototype, both in terms of improving the existing functionalities, and introducing new functionalities. The findings were examined with respect to performance and implementation complexity. Additionally, they were categorized by whether they are globally useful, only apply to specific simulation scenarios, or were found to not offer noteworthy improvements. Section 9.3 compiled a list of first-priority improvements that have reasonable implementation complexity, and significantly enhance the accelerator performance in practically any scenario. A package of improvements was grouped into a second priority level, which

mostly benefits Quantum Computing simulation usecases, and potentially Quantum Network simulators that allow OpenQASM back end integration.

8.2. Main Contributions

Chapter 1 formulated the question whether arbitrary quantum computation simulation can benefit from hardware acceleration on a parallelized, DDR-based computing platform. This thesis demonstrated how a tailor-made computation scheme for quantum state vectors can enable leveraging these platforms for simulating quantum operations, and proved the feasibility by implementing an operational prototype on an HPC FPGA. However, quantum gate operations with non-sparse gate matrices (*dense gates* in this thesis, see Section 2.3.1) proved to pose a performance limitation that is only partially solved in this thesis. Nevertheless, Section 6.3.6 underlined that a fully scaled-up future iteration of the implemented prototype can offer average speedup factors of up to 90, over a representative software baseline on an HPC CPU node. For large quantum states, the speedup depends on the performance of the explorative approach for dense gate operation matrices, that is proposed in Section 9.1.

This thesis yielded the following main contributions:

- novel *labeled-random* quantum state vector computation scheme
  - well suited for parallelized computation and DRAM or HBM operation
  - low memory footprint, due to expressing quantum operations as parallelizable per-element formulas, instead of compiling a gate operation matrix
  - full quantum operation functionality, including measurements and tensor operations (merging two quantum state vectors)
- prototype design and implementation
  - implementation of a functional, scalable quantum computation accelerator for arbitrary quantum operations, on a Data Center FPGA accelerator system
  - functionality support for both Quantum Computing and Quantum Network simulation
  - purely on-chip quantum state memory, including on-chip memory management
  - prototype speedups between factor 2 and 20 for quantum operations other than *dense gates*, compared to (single-thread) HPC CPU node
  - fully thought-out design extensions and optimizations for an extrapolated average speedup factor of roughly 90 (depending on simulation scenario)
  - modular design, compatible with different types of FPGA memory
  - generic host application software API, for integration into arbitrary quantum simulator front ends
  - portable, mostly platform-agnostic high-level design, allows for implementation on other accelerator platforms like GPUs
- auxiliary results
  - efficient, high-accuracy DSP-based FPGA implementation of numeric (inverse) square-root computation via Goldschmidt-Algorithm
  - evaluation of host-accelerator communication latencies on AMD Alveo Data Center FPGA cards

9 Future Work

The previous chapters derived and evaluated an accelerator prototype, followed by a discussion of performance improvements and functional additions within the current system architecture. This chapter highlights functionality additions that complement the existing system architecture, either in terms of functional modules, in terms of computing paradigm, or in terms of computation scheme. All of these approaches remain in a conceptual state, in contrast to the thought-out techniques in Chapter 7.

The subsequent discussion is structured as follows. Section 9.1 describes an alternative approach towards dense gate computation, which trades off small-state efficiency for linear scaling behavior. Section 9.2 explores potential future functional additions, which have yet to be thoroughly elaborated.

9.1. Linear-Scaling Dense Gate Computation

Like Section 7.2.5 earlier, this section addresses the accelerator's enormous performance disadvantages at dense gate computation, compared to the software baseline (Section 6.3.3). In contrast to Section 7.2.5, here a fundamentally different approach is outlined, with the focus on eliminating quadratic scaling behavior. The intention is not to replace the proposed extended dense gate array, but rather to complement it, for creating a hybrid in-hardware solution. The reason is that, due to drawbacks in parallelizability and memory access patterns of the techniques described in this section, the extended array as proposed in Section 7.2.5 is expected to perform drastically better up to a certain state size, namely as long as it is not dominated yet by scaling issues.

Section 7.2.5 explained that, even with maximum BRAM buffering, array-based dense gate computation is expected to perform worse than the software baseline for state sizes larger than 27 qubits. Both HBM stacks combined however can hold a state of up to 30 qubits (with leaving one PC group free for double-buffering between pre- and post-operation state, see Section 7.1.2). Therefore, it is desirable to conceive an alternative dense gate implementation, that, for smaller states, may even perform worse than an array, but can better handle these large states. This intended hardware design must match the linear scaling behavior of the software baseline. Translated to an operation scheme, the implication is that the alternative approach must rely on a fixed number of read and write iterations of the input and output states. Different from the BRAM-aided dense gate array, this section is limited to sketching a starting point for future evaluation, instead of a ready-to-implement in-detail design. It is demonstrated how pairwise gate operation can again be leveraged for a more efficient computation.

The proposed approach begins with abandoning fully sequential memory accesses - which in return allows storing partial results, and accessing them at a later point via their label. In that sense, this approach has similarities to the per-level BRAM buffer in the extended array approach. The proposal consists in using the main memory as both destination and buffer at the same time, instead of local BRAM buffers (which are not available in a sufficient quantity). Input vector elements are read sequentially. For every input vector element k_1, both its contributions to output vector elements k_1^*, k_2^* ($k_1 = k_1^*$) are computed in parallel (in the same way that the array's *label analyzer* and DSP multiplier do, but with two DSPs). Afterwards, the pair of partial results is written to one random-accessed destination memory location, which is determined by the smaller label's value. Once the input vector element k_2 is encountered ($k_2 = k_2^*$) and its partial output vector contributions are computed, the pair k_1^*, k_2^* is read

from main memory via random access, the new partial results added, and the now final output vector elements written back to the same memory location. Therefore, thanks to pairwise gate operation, two input vector elements only cause one ”read-add-write back” operation, instead of two.

It is evident that it is impossible for the proposed scheme to operate anywhere near the maximum HBM throughput. The presented concept causes constant random accesses across entire HBM PCs, with alternating read and write accesses. In addition, a comparison with the baseline's sparse matrix multiplication, in terms of required memory accesses per vector element, reveals a second drawback. When applying pairwise gate operation, sparse matrix multiplication consists of one random-access read, and one random-access write to a different location, per state vector entry. The proposed scheme needs one sequential read, one random-access read, and one random-access write to the same location - thus an additional read operation, and a memory controller bus turnaround. Lastly, data dependencies dictate that parallel processing of multiple elements, beyond a relatively low number, is likely to induce substantial routing complexity, because any processed input vector element can require an access to any output vector memory location.

In the context of the aforementioned drawbacks, it is important to point out that the predominant intention is to prevent the accelerator from losing any usefulness at large states, solely due to dense gate operations - not necessarily to outperform the software baseline. As the extrapolation in Section 6.3.6 demonstrates, even a dense gate implementation that is just “on par” with the baseline performance can allow for substantial over acceleration factors. It remains for future study to gather empirical data on the impact of the described memory access pattern, in order to establish a hardware performance projection for the proposed approach. Depending on the results, further exploration can be carried out into either optimizing the proposed scheme, or altering it into a more promising approach.

9.2. Exploration

9.2.1. FPGA Cluster

Section 5.1.1 discussed relevant characteristics for a target FPGA platform. The ability to perform peer-to-peer communication between accelerator nodes was named as one desirable asset, since it potentially enhances the usability of a device cluster. This section elaborates on the viability and impact of a multi-device implementation, both with and without peer-to-peer communication. In particular, two approaches are discussed, one without peer-to-peer linking, one with peer-to-peer linking. Although the discussion already has some level of technical depth, the topic remains considered “future work”, because a thorough exploration of the design options has not been conducted. Rather, this section is intended to provide a starting point for working out full-fledged approaches with and without peer-to-peer linking.

Without Peer-to-Peer Linking

The first paragraph concerns the situation without peer-to-peer linking. In that case, several accelerators would simply operate independently from each other, driven by the same host. This scenario is a good fit with the probabilistic nature of quantum computation simulations. It is common to execute a particular simulation a number of times, and study average results. A multi-core CPU can distribute parallel runs across different cores. However, the runtime does not fully scale with the number of cores, because the cores share cache memory, to an extent, and of course also the main memory connection. As Section 6.3.3 demonstrated, the amount of available cache memory has a huge impact on CPU performance. Parallel accelerators are not affected by that problem, since they do not share any memory infrastructure, and because in many situations there is only control communication between host and accelerator, but no data exchange. Therefore, operating several (unrelated) accelerators in parallel is expected to be a useful and practical setting, due to significantly better scaling behavior, compared to a highly parallelized CPU.

With Peer-to-Peer Linking

Considering peer-to-peer linking, it remains to be studied whether it can offer significant advantages, such as enhancing processing capabilities or enabling novel usecases. The following paragraph explores one conceivable usecases. However, it needs to be stated that the dataflow pattern of quantum computation simulations does not lend itself towards peer-to-peer clustering in the way that for instance activations do in neural networks, because quantum computation does not have a relatively small set of data that “propagates” through the computation.

A potential application that comes to mind is using a second accelerator as a quantum state memory and computation engine extension, where quantum states can be distributed across the memories on both devices. One instruction would be executed using two accelerators. In terms of theoretical feasibility, dense gates pose the only substantial problem. In monomial gates, state vector elements are processed one-by-one, with no dependencies. For measurements, the only point of synchronization, or dependency, is accumulating two partial expectation values. Tensor operations require no more inter-device data traffic than the batches of locally buffered state elements (*state 2* in Section 4.3.2). Dense gates however have data dependencies in the form of pairs of vector elements which appear at random, in an unordered vector, and therefore require data exchange via the serial link. The link, at $100\,\mathrm{Gbit\,s^{-1}}$, has 18 times less bandwidth than a 16-channel $450\,\mathrm{MHz}$ HBM read interface, and therefore poses a bottleneck. The combination with a random access-based memory approach for large dense gate operations (as motivated in Section 9.1) may still prove useful, because these computation schemes are considered unable to use the full memory bandwidth, and therefore would not be bottlenecked.

Despite the discussed usecase, one needs to realize that, due to exponential state vector size scaling, this “enhanced accelerator” architecture would only increase the maximum state size by 1 qubit - while smaller states are unlikely to require distribution across multiple memories. If a state is not distributed across multiple devices, then there is no benefit in having multiple devices connected, because one device would remain idle, while the other one is computing. In theory, two states can be processed at the same time, by always employing both device’s computation engine. That however would regularly require transferring one state from one platform to the other one, via a link that bottlenecks the quantum state memory interface. Concluding, there currently is no clear path for the design to benefit from peer-to-peer linking. Employing multiple devices appears to be done significantly more efficiently in the form of unrelated, parallel accelerators, as was described earlier in this section.

9.2.2. Density Matrices

In the scope of this thesis, labeled-random quantum state computation was only derived and formulated for the state vector formalism. It appears useful and viable to extend the scheme to density matrices (briefly introduced in Section 2.3.2), since these are computationally similar to state vectors. Furthermore, density matrices are widely used in Quantum Network simulation, which the presented accelerator is able to carry out, in contrast to existing hardware quantum computation implementations.

One challenge in integrating density matrices lies in enabling their great ability to reflect noisy operation, which is closely linked to mixed states. The current computation engine is not equipped to perform any operation that creates a mixed state, which applies to any of the “discrete action” error models described in Section 2.3.2 (relaxation, dephasing, depolarization, bit-flipping). However, since the primary reason to use density matrices is error modeling, or noise modeling, an implementation without the functionality to handle mixed states is of little to no use. Therefore, the computation engine needs to be extended accordingly, and with it the entire instruction handling chain from front end interface, through low-level software and the intermediary layer register file, to additional FSMs in the quantum state module’s control block (see Figure 4.7).

Another important aspect in supporting density matrices lies in how to handle element labels. Theoretically, the labels need to be two-dimensional, which uses up significantly more memory per matrix element. An alternative approach to two-dimensional labels is exploiting the regular matrix structure, by “encoding” one dimension in the aligned memory address. It might be mathematically possible to establish the design rule that the matrix is always stored row-wise or column-wise, without compromising parallel operation and sequential memory accessing. In that case, if for instance a matrix is known to be 2x2, with row-wise elements in consecutive memory that take up $128\,\mathrm{bit}$ each, it can be determined that the two elements starting at address 0 belong to one row, and the two elements starting at address 8 (in bytes) belong to the other row. A brief example analysis of an X gate, applied to a 2x2 (1-qubit) density matrix, motivates this approach:

$$\begin{aligned} X \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot X &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot X \\ &= \begin{pmatrix} c & d \\ a & b \end{pmatrix} \cdot X \\ &= \begin{pmatrix} d & c \\ b & a \end{pmatrix} \end{aligned} \tag{9.1}$$

Evidently, this computation can generally be carried out with sequential accesses by storing the input and output matrices row-wise, with one-dimensional labeled-random elements per row. However, since the first row becomes the second row and vice versa, the information on the order of rows needs to be stored as well.

Equation (9.1) can provide a more compact scheme than full two-dimensional labels, provided that future study proves its applicability to all supported quantum operations. Careful consideration on how to store the order of matrix rows appears to be advisable. The row indices should be stored together as one package, such that they do not interrupt consecutive reading of matrix elements, or row alignment in memory (if they were to be stored interleaved with the respective rows). It might even be beneficial to use an entirely different physical memory, like a portion of UltraRAM, in order to preserve proper address alignment of states in state memory at all times.

It remains for future study to transfer labeled-random computation to density matrices, based on the motivation and challenges formulated in this section. First, the element update formulas need to be adapted for density matrices. Second, the engine's functionality set needs to be adapted as described in this section. Third, the most efficient approach for two-dimensional labels must be determined, and implemented. In conclusion, the projection is that support for density matrices can greatly boost the accelerator's usefulness, appears to be feasible, but comes at a certain level of design and implementation complexity.

9.3. Conclusion

This chapter examined potential avenues for future functionality extension. While conceptual approaches are presented, most of the presented additions require further investigation, before making a decision on whether to integrate them in the proposed form.

Section 9.1 discussed an approach towards a computation core for dense gates with linear computation time scaling (in the state vector size), compared to quadratic scaling with the currently present dense gate array. The applicability of the presented scheme remains subject to further quantitative studies into HBM random-access latencies, and further development towards parallelizability of the approach. Nonetheless, it appears evident that the module would exist in a hybrid solution with an improved dense gate array (as proposed in Section 7.2.5), because the array is highly likely to perform better up to state sizes of at least 20 qubits.

Section 9.2 elaborated on functionality additions that remain subject to exploration. Firstly, the possibility to use the accelerator in an FPGA cluster was studied (Section 9.2.1). It was concluded that the most effective use of multiple devices lies in distributing probabilistic iterations of a simulation across parallel, unrelated accelerators - due to significantly better scaling behavior than parallel iterations on a multi-core CPU. However, no clear path could be found towards efficiently utilizing peer-to-peer accelerator connectivity, which has its main cause in unfavorable dataflow patterns. Secondly, Section 9.2.2 collected and assessed the necessary design additions to support density matrix computation, as an alternative to state vectors. It was found that the integration requires an adaptation of the computation scheme, and a new computation engine module, but appears to be feasible. Given the accelerator's already existing ability to handle Quantum Network simulations, and the manageable implementation complexity, an effort towards supporting density matrix is considered a reasonable use of development resources, and should receive a high priority.

References

[1] Andrew Boutros and Vaughn Betz. “FPGA Architecture: Principles and Progression”. In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. issn: 1558-0830. doi: 10.1109/MCAS.2021.3071607. url: https://ieeexplore.ieee.org/abstract/document/9439568 (visited on 07/28/2025).

[2] *UltraScale Architecture Configurable Logic Block - UG574 (v1.6)*. Jan. 22, 2025. url: https://docs.amd.com/r/en-US/ug574-ultrascale-clb?tocId=2DOfkt0abl3oJAYunYLu0A (visited on 12/07/2025).

[3] *UltraScale Architecture Memory Resources - UG573 (v1.14)*. Nov. 18, 2025. url: https://docs.amd.com/r/en-US/ug573-ultrascale-memory-resources/Introduction-to-the-UltraScale-Architecture (visited on 12/07/2025).

[4] *Vitis Unified Software Platform Documentation - UG1393 (v2022.2)*. May 25, 2022. url: https://docs.amd.com/r/2022.1-English/ug1393-vitis-application-acceleration/Platform (visited on 12/07/2025).

[5] *UltraScale Architecture DSP Slice - UG579 (v1.11)*. Aug. 30, 2021. url: https://docs.amd.com/v/u/en-US/ug579-ultrascale-dsp (visited on 12/07/2025).

[6] *Alveo Data Center Accelerator Platforms - UG1120 (v2.0.1)*. Oct. 11, 2023. url: https://docs.amd.com/r/en-US/ug1120-alveo-platforms/Overview (visited on 12/07/2025).

[7] *FPGA Acceleration of Matrix Multiplication for Neural Networks - XAPP 1332 (v1.0)*. Feb. 27, 2020. url: https://docs.amd.com/v/u/en-US/xapp1332-neural-networks (visited on 12/07/2025).

[8] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. 10th anniversary edition. Cambridge: Cambridge university press, 2010. isbn: 978-1-107-00217-3.

[9] P.W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 35th Annual Symposium on Foundations of Computer Science. Nov. 1994, pp. 124–134. doi: 10.1109/SFCS.1994.365700. url: https://ieeexplore.ieee.org/abstract/document/365700 (visited on 11/27/2025).

[10] Lov K. Grover. “A fast quantum mechanical algorithm for database search”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC ’96*. the twenty-eighth annual ACM symposium. Philadelphia, Pennsylvania, United States: ACM Press, 1996, pp. 212–219. isbn: 978-0-89791-785-8. doi: 10.1145/237814.237866. url: http://portal.acm.org/citation.cfm?doid=237814.237866 (visited on 11/27/2025).

[11] Davide Castelvecchi. “IBM releases first-ever 1,000-qubit quantum chip”. In: *Nature* 624.7991 (Dec. 4, 2023). Bandiera_abtest: a Cg_type: News Publisher: Nature Publishing Group Subject_term: Mathematics and computing, Quantum physics, pp. 238–238. doi: 10.1038/d41586-023-03854-1. url: https://www.nature.com/articles/d41586-023-03854-1 (visited on 11/27/2025).

[12] Arian J. Stolk et al. “Metropolitan-scale heralded entanglement of solid-state qubits”. In: *Science Advances* 10.44 (Oct. 30, 2024). Publisher: American Association for the Advancement of Science, eadp6442. doi: 10.1126/sciadv.adp6442. url: https://www.science.org/doi/full/10.1126/sciadv.adp6442 (visited on 11/27/2025).

[13] Tim Coopmans et al. “NetSquid, a NETwork Simulator for QUantum Information using Discrete events”. In: *Communications Physics* 4.1 (July 16, 2021). Publisher: Nature Publishing Group, pp. 1–15. issn: 2399-3650. doi: 10.1038/s42005-021-00647-8. (Visited on 04/21/2024).

[14] Xiaoliang Wu et al. “SeQUeNCe: a customizable discrete-event simulator of quantum networks”. In: *Quantum Science and Technology* 6.4 (Sept. 2021). Publisher: IOP Publishing, p. 045027. issn: 2058-9565. doi: 10.1088/2058-9565/ac22f6. url: https://dx.doi.org/10.1088/2058-9565/ac22f6 (visited on 04/08/2024).

[15] Stephen DiAdamo et al. “Packet switching in quantum networks: A path to the quantum Internet”. In: *Physical Review Research* 4.4 (Oct. 28, 2022). Publisher: American Physical Society, p. 043064. doi: 10.1103/PhysRevResearch.4.043064. url: https://link.aps.org/doi/10.1103/PhysRevResearch.4.043064 (visited on 04/05/2024).

[16] Ryosuke Satoh et al. “QuISP: a Quantum Internet Simulation Package”. In: *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2022 IEEE International Conference on Quantum Computing and Engineering (QCE). Sept. 2022, pp. 353–364. doi: 10.1109/QCE53715.2022.00056. url: http://ieeexplore.ieee.org/abstract/document/9951186?casa_token=ewuPoR4ywIQAAAAA:c_7fVKpw77qpSeJnIqGDK-jQ2HHFYf9ViVg5fD9M6ceKCiINQsnn6LH420dI-KTDjnqUi6NC (visited on 04/08/2024).

[17] Tyson Jones et al. “QuEST and High Performance Simulation of Quantum Computers”. In: *Scientific Reports* 9.1 (July 24, 2019). Publisher: Nature Publishing Group, p. 10736. issn: 2045-2322. doi: 10.1038/s41598-019-47174-9. url: https://www.nature.com/articles/s41598-019-47174-9 (visited on 09/08/2025).

[18] Samuel Jaques and Thomas Häner. “Leveraging State Sparsity for More Efficient Quantum Simulations”. In: *ACM Transactions on Quantum Computing* 3.3 (June 30, 2022), 15:1–15:17. doi: 10.1145/3491248. url: https://dl.acm.org/doi/10.1145/3491248 (visited on 09/08/2025).

[19] *QuTech*. TU Delft. url: https://www.tudelft.nl/en/qutech (visited on 01/09/2026).

[20] *TNO*. tno.nl/en. url: https://www.tno.nl/en/ (visited on 01/09/2026).

[21] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020). Publisher: Nature Publishing Group, pp. 357–362. issn: 1476-4687. doi: 10.1038/s41586-020-2649-2. url: https://www.nature.com/articles/s41586-020-2649-2 (visited on 12/02/2025).

[22] Stefan Behnel et al. “Cython: The Best of Both Worlds”. In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 31–39. issn: 1558-366X. doi: 10.1109/MCSE.2010.118. url: https://ieeexplore.ieee.org/abstract/document/5582062 (visited on 12/02/2025).

[23] Agostino Giorgio. “Project and Implementation of a Quantum Logic Gate Emulator on FPGA Using a Model-Based Design Approach”. In: *IEEE Access* 12 (2024), pp. 41317–41353. issn: 2169-3536. doi: 10.1109/ACCESS.2024.3377458. url: https://ieeexplore.ieee.org/abstract/document/10472476 (visited on 11/04/2025).

[24] Naveed Mahmud and Esam El-Araby. “A Scalable High-Precision and High-Throughput Architecture for Emulation of Quantum Algorithms”. In: *2018 31st IEEE International System-on-Chip Conference (SOCC)*. 2018 31st IEEE International System-on-Chip Conference (SOCC). ISSN: 2164-1706. Sept. 2018, pp. 206–212. doi: 10.1109/SOCC.2018.8618545. url: http://ieeexplore.ieee.org/abstract/document/8618545?casa_token=ScBL3GFj_wAAAAAA:7aGIr7vkIshvuOW6VDwvPjfRjF3bvc8pVmrBWUZCVH1p3wum7_dUIZ_zyWp_ErdqC34EOrOe (visited on 04/09/2024).

[25] Yunpyo Hong et al. “Quantum Circuit Simulator based on FPGA”. In: *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*. 2022 13th International Conference on Information and Communication Technology Convergence (ICTC). ISSN: 2162-1241. Oct. 2022, pp. 1909–1911. doi: 10.1109/ICTC55196.2022.9952408. url: http://ieeexplore.ieee.org/abstract/document/9952408?casa_token=ei6Z2d5qZ0gAAAAA:m2C-F7LGc-Gl8a-UqaOmS1gOgW7hVFMWHAGA8YdqQbXir7YUl4NdLmX12WRU5c3hgcAQMyB_ (visited on 04/09/2024).

[26] A.U. Khalid, Z. Zilic, and K. Radecka. “FPGA emulation of quantum circuits”. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. ISSN: 1063-6404. Oct. 2004, pp. 310–315. doi: 10.1109/ICCD.2004.1347938. url: http://ieeexplore.ieee.org/abstract/document/1347938?casa_token=XE9Ho-H1adMAAAAA:HUDhTWodBeBqeXUc9iSKg6wiNhr0qWrVwBzZrz_HnXJYcGBkacIdAcBSNWMn39svKEBFzPpg (visited on 04/09/2024).

[27] Agustin Silva and Omar Gustavo Zabaleta. “FPGA quantum computing emulator using high level design tools”. In: *2017 Eight Argentine Symposium and Conference on Embedded Systems (CASE)*. 2017 Eight Argentine Symposium and Conference on Embedded Systems (CASE). Aug. 2017, pp. 1–6. doi: 10.23919/SASE-CASE.2017.8115369. url: http://ieeexplore.ieee.org/abstract/document/8115369 (visited on 04/09/2024).

[28] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono. “An FPGA-Based Quantum Computing Emulation Framework Based on Serial-Parallel Architecture”. In: *International Journal of Reconfigurable Computing* 2016 (Apr. 7, 2016). Publisher: Hindawi, e5718124. issn: 1687-7195. doi: 10.1155/2016/5718124. url: https://www.hindawi.com/journals/ijrc/2016/5718124/ (visited on 04/09/2024).

[29] Zeke Wang et al. “Shuhai: Benchmarking High Bandwidth Memory On FPGAS”. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). ISSN: 2576-2621. May 2020, pp. 111–119. doi: 10.1109/FCCM48280.2020.00024. url: https://ieeexplore.ieee.org/document/9114755 (visited on 06/19/2024).

[30] Bernhard Burgstaller and Friedrich Pillichshammer. “THE AVERAGE DISTANCE BETWEEN TWO POINTS”. In: *Bulletin of the Australian Mathematical Society* 80.3 (Dec. 2009), pp. 353–359. issn: 1755-1633, 0004-9727. doi: 10.1017/S0004972709000707. url: https://www.cambridge.org/core/journals/bulletin-of-the-australian-mathematical-society/article/average-distance-between-two-points/F182A617B5EC6DB5AD31042A4BDF83AE (visited on 12/01/2025).

[31] Longshan Xu et al. “Accelerating Quantum Circuit Simulations with Data Compression”. In: *Advanced Quantum Technologies* n/a (n/a). _eprint: https://advanced.onlinelibrary.wiley.com/doi/pdf/10.1002/qute.202500223, e2500223. issn: 2511-9044. doi: 10.1002/qute.202500223. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/qute.202500223 (visited on 09/08/2025).

[32] Mason Woo et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. 3rd. USA: Addison-Wesley Longman Publishing Co., Inc., July 1999. isbn: 978-0-201-60458-0.

[33] John Nickolls et al. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (Mar. 2008). Publisher: Association for Computing Machinery, pp. 40–53. doi: 10.1145/1365490.1365500. url: https://spawn-queue.acm.org/doi/full/10.1145/1365490.1365500 (visited on 11/30/2025).

[34] Robert E. Goldschmidt. “Applications of division by convergence”. Accepted: 2005-08-17T23:29:41Z. Thesis. Massachusetts Institute of Technology, 1964. url: https://dspace.mit.edu/handle/1721.1/11113 (visited on 11/30/2025).

[35] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, July 2007. isbn: 978-0-321-51526-1.

[36] *Alveo U55C Data Center Accelerator Cards Data Sheet - DS978 (v1.3)*. June 23, 2023. url: https://docs.amd.com/r/en-US/ds978-u55c/Product-Details (visited on 12/07/2025).

[37] *AXI High Bandwidth Memory Controller - PG276 (v1.0)*. Nov. 18, 2024. url: https://docs.amd.com/r/en-US/pg276-axi-hbm/Introduction (visited on 12/07/2025).

[38] *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs - UG949 (v2022.2)*. Nov. 30, 2022. url: https://docs.amd.com/r/2022.2-English/ug949-vivado-design-methodology/Introduction (visited on 12/07/2025).

[39] *Efficient Shift Registers, LFSR Counters, and Long PseudoRandom Sequence Generators - XAPP 052 (v1.1)*. July 7, 1996. url: https://docs.amd.com/v/u/en-US/xapp052 (visited on 12/07/2025).

[40] Javier Moya et al. “fpgasystems/hacc: ETHZ-HACC”. In: Zenodo, Sept. 2023. doi: 10.5281/zenodo.8340448. url: https://doi.org/10.5281/zenodo.8340448.

[41] Austin G. Fowler et al. “Surface Code Quantum Communication”. In: *Physical Review Letters* 104.18 (May 6, 2010), p. 180503. issn: 0031-9007, 1079-7114. doi: 10.1103/PhysRevLett.104.180503. url: https://link.aps.org/doi/10.1103/PhysRevLett.104.180503 (visited on 11/27/2025).

[42] Tianjie Hu, Jindi Wu, and Qun Li. “Quantum Network Routing Based on Surface Code Error Correction”. In: *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. 2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS). ISSN: 2575-8411. July 2024, pp. 1236–1247. doi: 10.1109/ICDCS60910.2024.00117. url: https://ieeexplore.ieee.org/abstract/document/10631008 (visited on 11/27/2025).

[43] K.M. Svore et al. “A layered software architecture for quantum computing design tools”. In: *Computer* 39.1 (Jan. 2006), pp. 74–83. issn: 1558-0814. doi: 10.1109/MC.2006.4. url: https://ieeexplore.ieee.org/abstract/document/1580386 (visited on 10/28/2025).

[44] Andrew W. Cross et al. *Open Quantum Assembly Language*. July 13, 2017. doi: 10.48550/arXiv.1707.03429. arXiv: 1707.03429[quant-ph]. url: http://arxiv.org/abs/1707.03429 (visited on 10/28/2025).

[45] Andrew Cross et al. “OpenQASM 3: A Broader and Deeper Quantum Assembly Language”. In: *ACM Transactions on Quantum Computing* 3.3 (Sept. 6, 2022), 12:1–12:50. doi: 10.1145/3505636. url: https://dl.acm.org/doi/10.1145/3505636 (visited on 10/28/2025).

[46] Axel Dahlberg et al. “NetQASM—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet”. In: *Quantum Science and Technology* 7.3 (June 2022). Publisher: IOP Publishing, p. 035023. issn: 2058-9565. doi: 10.1088/2058-9565/ac753f. url: https://dx.doi.org/10.1088/2058-9565/ac753f (visited on 04/22/2024).

[47] *IBM Quantum Computing | Qiskit*. url: https://www.ibm.com/quantum/qiskit (visited on 10/28/2025).

Acronyms

AI	Artificial Intelligence
ALU	Arithmetical-Logical Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CAM	Content-Addressable Memory
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CMT	Clock-Management Tile
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CU	Compute Unit
DDR	Double Data Rate
DLL	Dynamic Link Library
DMA	Direct Memory Access
DRAM	Dynamic RAM
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit
GT	Gigabit Transceiver
HACC	Heterogeneous Accelerated Compute Cluster
HBM	High-Bandwidth Memory
HLS	High-Level Synthesis
HPC	High-Performance Computing
ISA	Instruction Set Architecture
LFSR	Linear-Feedback Shift Register
LUT	Look-up Table
LUTRAM	Look-up Table Random-Access Memory
MC	Memory Controller
ML	Machine Learning
PE	Processing Element
RAM	Random-Access Memory
RNG	Random Number Generator
RTL	Register-Transfer Level
QFT	Quantum Fourier Transform

QKD Quantum Key Distribution
PC Pseudo-Channel
PnR Place & Route
PRNG Pseudo Random Number Generator
SDP Simple Dual-Port
SIMD Single-Instruction Multiple-Data
SLR Super-Logic Region
SLL Super-Long Line
SoC System on Chip
SRAM Static RAM
SSI Stacked Silicon Interconnect
TDP True Dual-Port
TRNG True Random Number Generator
XPM Xilinx Programming Macro
XRT Xilinx Runtime Library

Glossary

This thesis introduces several terms that are unique to this thesis, in order precisely point out specific aspects, or due to considerable reuse. These terms are listed in the following, briefly explained, and linked to their respective first appearance (and definition) in the thesis.

Quantum Computation defined in: Section 2.3.3
Quantum Computation serves as a collective term for any computational application based on Quantum Information - among which Quantum Computing and Quantum Networks.

(Quantum) State Vector Computation defined in: Section 2.4.2
(Quantum) State Vector Computation refers to any mathematical operation on quantum state vectors. It serves as a generic term for various implementations or computation schemes, within the state vector formalism.

Base Gate Matrix/Full Operation Matrix defined in: Section 2.3.1
The *base gate matrix* is the 2^nx2^n operation matrix for an n-qubit quantum gate U (2x2 for single-qubit gates). The *full operation matrix* is the 2^mx2^m matrix when applying U to one or more qubits in a quantum system with $m \geq n$ qubits, that in most cases is obtained by performing tensor products between U and identity matrices.

Monomial/Non-monomial/Sparse/Dense Gates defined in: Section 2.3.1
Monomial or *sparse* quantum gates are quantum gates with monomial gate matrices (monomial matrices are matrices with exactly one non-zero entry per row and column). *Non-monomial* or *dense* quantum gates are gates with one target qubit, an arbitrary number of control qubits (applied to a quantum system of arbitrary size) with non-monomial gate matrices. They are treated as having exactly two non-zero entries per row and column, although the actual number might be lower, but it is never higher.

Diagonal/Antidiagonal Gates defined in: Section 3.5.1
Diagonal gates are quantum gates with non-zero elements in their gate operation matrix only along the main diagonal. *Antidiagonal* gates are quantum gates with non-zero elements in their gate operation matrix only along the anti-diagonal (the diagonal going from the lower left to the upper right corner).

Pairwise Gate Operation defined in: Section 2.4.2
Pairwise gate operation refers to performing single-qubit gate operations on quantum state vectors of arbitrary size, by applying a 2x2 base gate operation matrix to pairs of input state vector elements as individual multiplications with 2-element vectors.

Labeled-Hashed Quantum State Vector Representation defined in: Section 2.4.2
Labeled-hashed quantum state vector computation refers to the quantum state vector representation introduced by Jaques. et al [18], that stores state vector elements as label-amplitude pairs, with hashed memory addresses.

Standard Complex Multiplier defined in: Section 5.4.1
The *standard complex multiplier* is the elementary DSP-based FPGA multiplication circuit, that is used by most of the computation engine's operation module PEs.

Engine Data Bus defined in: Section 5.4.2
The *engine data bus* is the custom, dual-simplex data bus in the FPGA prototype implementation, that connects the computation engine to the quantum state memory back end.

Acknowledgements

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Cluster (HACC) program [40].

For this work, the NetSquid development team kindly provided full source code access.

For their technical support during the thesis work, I want to thank Mark van Beusekom from TU Delft's QCE department, who helped me numerous times with the institute's server infrastructure, and Michał van Hooft from QuTech, who connected me with the NetSquid development team, and provided me access to QuTech's Qraken server for the software baseline experiments.

I want to thank the D.S.J.V. Groover student association and the Groover Big Band, for creating a welcoming environment from the start, for many memorable moments, for the trust that was put in me during my committee work, and for trying their best to teach me being a little less serious from time to time.

Anti-Acknowledgements

During my studies at TU Delft, next to people who were helpful to me or inspired me, I have unfortunately encountered more situations than I find acceptable that to me conveyed a mixture of carelessness, complacency, and lack of respect, towards me as a student. I have decided to include several of these in my thesis, because they appear to follow a pattern, rather than being individual incidents, and because they significantly ruptured my enthusiasm for learning and my fun with working on Engineering problems, during my time at the university.

I have experienced the following situations first-hand, in the context of core subjects of the Computer Engineering programme, if not stated otherwise. Statements reflect my subjective opinions.

On TU Delft's department of Computer Engineering

As a professor, you should not tell your students that you “don’t read their reports anyways”, you “just request the reports to make sure that [the students] get started”. Nor should you act this way in the first place, obviously.

Oral group exams with individual grades, in a way that forces the students to compete against each other live during the exam, are an expression of a toxic academic culture. The format should be prohibited.

A course is not supposed to have the predominant character of a promotion event for the lecturer’s research project.

Grading students on peer reviewing each other’s reports, without providing feedback, is at least as much delegating the lecturer’s task to the students, as it is an exercise.

I was mocked (instead of properly advised) by a professor for “re-inventing” a common technique (which ten minutes later they understood I had not) and for imprecise use of technical vocabulary. They were not even familiar with the underlying subject.

If an entire class of students is consistently using the same terms in the same incorrect way, that the same lecturer has introduced to that class in the quarter before, it is at least questionable for the lecturer to openly call out the students on whether they have “by any chance followed [said course] last quarter” - and inappropriate to do so in a sarcastic tone.

It was an almost consistent observation that, whenever I took courses outside of the Computer Engineering department, the teaching quality significantly improved (presentation, course structure, exam design), and with it the sense of mutual respect in the interaction between student and lecturer. For several core Computer Engineering courses however, the established practice among students for effective learning was - and had to be - to exclusively rely on external lecture recordings on the same subject. As a positive counterexample at the other end of the scale, I want to name the courses given under the roof of QuTech, which eventually sparked inspiration for this thesis work.

Various events showed a pattern of the department expecting standards from its students, which it clearly failed to fulfill itself.

- My studies included a mandatory class on academic presenting, which I find reasonable. Yet a large number of the programme’s core lectures showed not even an attempt to adhere to high presentation standards themselves - referring to both aural and visual presentation. They rather conveyed a recurring sense of indifference towards the students, while students are being graded on presentation quality.

- Mails and messages to professors regularly remained unanswered or overlooked, even on important matters. I was told about one professor that they are “not really a message person”, as if it was normal to expect me to adapt to that - while other staff might be only reachable via message, and not via mail, or exclusively in person. Yet I as a student am (understandably) expected to be on top of **all** official communication channels, and a failure in doing so is considered my own problem. The same standards should apply to the academic staff. People with experience in the department called the incidents “normal” - in the sense of them being unsurprising and common, not in the sense of them being acceptable.
- As part of the application to the Computer Engineering programme, applicants had to submit GRE General Test scores - the most stressful, in my opinion inhumane, exam I have ever taken, which also added €270 to the application cost - and were expected to achieve relatively high scores. Additionally, an extensive motivation letter had to be handed in. Demanding this level of dedication and excellence, to then treat student experience with such negligence, is grotesquely detached, and grotesquely arrogant.

The justification to expect standards from others should be earned by setting the positive example, even between university and student.

On TU Delft as an organization

TU Delft’s published growth plans raise serious concerns about indifference towards both its students, and initially the residents of Delft as a municipality. In a situation of drastic housing shortage, in Delft and **all** surrounding cities, aiming for about $10{,}000$ additional students anywhere in the region ignores all students’ already terrible (and highly exploitable) housing situation, which TU Delft should instead aim to improve. Not even consulting the municipality of Delft, before announcing growth plans in the order of $10\,\%$ of the city’s total population, indicates a lack of prudence and awareness, in the best case.

The reaction by the TU Delft Executive Board to the Inspectorate of Education’s 2024 report on social security within the university showcases a ridiculous level of arrogance, that I hardly find words to describe. Threatening a lawsuit against the report’s authors (instead of seeking cooperation to address the concerns), and later on denying the official campus magazine full legal security in their press work, should have been more than sufficient reason for the board to step down immediately. The reaction suggests that the university’s reputation stands above everything, particularly above the interests and wellbeing of me as a student, or anyone else in a comparatively low position in the academic hierarchy.

Conclusion

When I started my master’s studies at TU Delft, I was excited about being accepted to the Computer Engineering programme, and I was looking forward to the studies and experiences in Delft. I graduate with the negative feelings outweighing the positive ones, with the impression of a department that has a far-reaching culture issue, and with the impression of an institution that is predominantly concerned with its image to the academic outside, instead of ensuring a quality environment to the inside.

A

Quantum Computing Fundamentals

A.1. Elementary Quantum States

A.1.1. Cardinal States

$|+\rangle$ / $|-\rangle$ and $|+i\rangle$ / $|-i\rangle$ can be expressed as superpositions of $|0\rangle$ / $|1\rangle$ as follows (note the necessary normalization factor $\frac{1}{\sqrt{2}}$:

$$|+\rangle \coloneqq \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\,, \qquad |-\rangle \coloneqq \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$
$$|+i\rangle \coloneqq \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)\,, \qquad |-i\rangle \coloneqq \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)$$

A.1.2. Bell States

The Bell states are a set of four specific fully-entangled 2-qubit states. The names Bell state and EPR pair are interchangeable.

$$\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right)$$
$$\left|\Phi^-\right\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle - |11\rangle\right)$$
$$\left|\Psi^+\right\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle + |10\rangle\right)$$
$$\left|\Psi^-\right\rangle = \frac{1}{\sqrt{2}}\left(|01\rangle - |10\rangle\right)$$

A.2. Elementary Quantum Gates

A.2.1. Single-Qubit Gates

Pauli-X (X) $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ Pauli-Y (Y) $\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$

Pauli-Z (Z) $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ Hadamard (H) $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Phase (S,P) $\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$ $\pi/8$ (T) $\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$

A.2.2. Controlled Gates

Controlled-X (CNOT)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
Controlled-Z (CZ)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$
Toffoli	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$

The 2x2 matrices assume that the first qubit (A in the example in Section 2.3.1) is the control qubit, and the second qubit (B) is the target qubit. The respective controlled single-qubit gate appears as a submatrix along the main diagonal of the controlled gate matrix, while the other submatrix along the main diagonal is an identity matrix. For the Toffoli gate, the first two qubits are control qubits, and the last qubit is the target qubit. Note that the Toffoli gate is essentially a CNOT gate, but with 2 control qubits instead of 1.

When swapping control and target qubit in a controlled 2-qubit gate, the identity and base gate submatrices become “stretched out” across the 2x2 matrix, as shown below:

CNOT_{BA}	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$
CZ_{BA}	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$

Conveniently, for the CZ gate, one obtains the same matrix.

A.2.3. Other Gates

SWAP	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Swaps the (partial) states of two qubits.

B

Mathematical Proofs

B.1. Row and Column Elements in Kronecker Product

This section underlines the following claim: **The result of a tensor product between an arbitrary matrix M and an n-dimensional identity matrix (in any order) is a matrix that consists of n instances of any row and column in M, with optionally inserted zero-elements, and only zero-elements otherwise.**

The proof can be indicated graphically, by looking at an example. The term *coincidence* is used in this elaboration to indicate that two elements within one matrix appear in the same row or column.

$$M \coloneqq \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\mathbb{I}_2 \otimes M = \begin{pmatrix} a & b & 0 & 0 \\ d & c & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix} \tag{B.1}$$

$$M \otimes \mathbb{I}_2 = \begin{pmatrix} a & 0 & b & 0 \\ 0 & a & 0 & b \\ c & 0 & d & 0 \\ 0 & c & 0 & d \end{pmatrix} \tag{B.2}$$

Even for higher matrix dimensions, the first case (Equation (B.1)) is trivial: M is always "duplicated" in a "non-overlapping" way (meaning there are no coincidences between the different "appearances" of M as a sub-matrix in $\mathbb{I}_2 \otimes M$). Therefore, the number of non-zero elements in any row and column is the same as in the corresponding row or column in M. The same argument can be applied to the second case (Equation (B.2)) as well: Here, the "appearances" of M are "stretched out" in the resulting matrix. Rows and columns 1 and 3 contain the first appearance (caused by the upper left 1 in $\mathbb{I}_2$), rows and columns 2 and 4 contain the second appearance. Again, any row or column has a corresponding row or column in M, and no coincidences are created, or removed. This also holds for a change to either matrix dimension, because the "stretching factor" scales accordingly. Both cases combined prove the above statement.

As an implication, tensor product with an identity matrix can be said to "preserve" the number of non-zero elements per matrix row and column.

B.1.1. Monomial Gate Extension

As an immediate consequence of the above proof, the tensor product between a monomial matrix and an identity matrix is always again a monomial matrix. Therefore, an extended full operation matrix for a quantum gate is monomial if and only if the base gate matrix is monomial.

C

Algebraic Element Update Formulas

Section 3.5 introduced element update formulas for all non-controlled gate operations, mentioning that a more algebraic set of equivalent formulas was formulated during an earlier phase of the thesis work. This applies to all element update formulas whose newer version is based on pairwise gate operation - namely, diagonal gates, antidiagonal gates, and dense gates. For these operations, this section provides the prior formulations, which at the time of writing are still used in any delivered implementation code and simulation scripts. It was not investigated to which extent the hardware synthesis tool manages to "reduce" the legacy formulations to the simpler new formulations, which would cause the "actual" implemented difference to become marginal.

C.1. Antidiagonal Gates

In this section, antidiagonal gates are discussed before diagonal gates, because the algebraic formula for antidiagonal gates is more complicated. Several parts of the argumentation translate to diagonal gates, as will be seen. The analysis is based on Figure C.1.

- **Label**

  1. **l-block**: one wants to perform a type of "binning" operation, where column indices k that fall into the same l-block are grouped together. Dividing k by the l-width, and rounding down the result to an integer, tells the "count" of the l-block (which l-block it is, along the main diagonal). The count, multiplied with the l-width, tells the block's "starting index" (i_l in the following), thus the first row index that the block occupies. As one term:
     $2^{r+1} \cdot \left\lfloor \frac{k}{2^{r+1}} \right\rfloor$.
     Because the blocks do not overlap, k^* is guaranteed to lie between i_l and $(i_l + \text{r-width} - 1)$.
  2. **r-block**: The concept of "binning" via division, rounding down, and subsequently multiplying, applies again - only this time with the r-width, instead of the l-width. The numerator however is different. Firstly, k is "normalized" to the width of an l-block by computing $k \bmod 2^{r+1}$, because it is not important anymore in which l-block k falls. Due to the antidiagonal structure, it is intuitive to deduct the normalized k from the highest index, which is again the l-width 2^{r+1}. Lastly, -1 translates from 1-indexing to 0-indexing, because 2^{r+1} as an index would be 1-indexed, while the binning method uses 0-indexing. Accumulated, that yields the second term, hence the "starting row index of the r-block, within the l-block":
     $2^r \cdot \left\lfloor \frac{2^{r+1} - (k \bmod 2^{r+1}) - 1}{2^r} \right\rfloor$
  3. **element**: The last term needs to determine the exact row index of the non-zero element in matrix column k, within the identified r-block. Since in this case, an r-block is a diagonal matrix, that is a straightforward computation. One simply needs to apply the "normalization" step from the r-block term, but with respect to the r-width:
     $k \bmod 2^r$

  All three terms accumulated yield the row index of the non-zero element in column k, or, in other words, the new element index k^*.

$$M := \begin{pmatrix} 0 & a \\ b & 0 \end{pmatrix}, \quad \mathbb{I}_2 \otimes M \otimes \mathbb{I}_2 = \begin{pmatrix} & & a & 0 & & \\ & & 0 & a & & \\ b & 0 & & & & \\ 0 & b & & & & \\ & & & & a & 0 \\ & & & & 0 & a \\ & & b & 0 & & \\ & & 0 & b & & \end{pmatrix}$$

$$k^* = 2^{r+1} \left\lfloor \frac{k}{2^{r+1}} \right\rfloor + 2^r \left\lfloor \frac{2^{r+1} - (k \bmod 2^{r+1}) - 1}{2^r} \right\rfloor + k \bmod 2^r$$

$$v^* : k \bmod 2^{r+1} \begin{cases} < 2^r & \rightarrow v^* = v \cdot b \\ \geq 2^r & \rightarrow v^* = v \cdot a \end{cases}$$

Figure C.1: derivation of the element update formula for antidiagonal gates (highlighting boxes for $k = 7$)

- **Element Factor**

  The procedure for obtaining the multiplicative factor is significantly more simple, in this case.

  1. **l-block**: Since every l-block is the exact same submatrix, it does not matter in which one k exactly falls. Therefore, the already introduced normalization step $k \bmod 2^{r+1}$ is applied, which "maps" any k into the first l-block, and the calculation proceeds based on that value.
  2. **r-block**: Within an l-block in this example, it is again not important which row the column $k \bmod 2^{r+1}$ exactly corresponds to (in terms of non-zero element). Looking at the l-block columnwise, one notices that all non-zero elements in the lower-indexed half, and in the higher-indexed half, are the same, respectively. Note that both halfs have exactly the width of an r-block. Therefore, it is sufficient for the formula to determine which half of the l-block $k \bmod 2^{r+1}$ falls into, which is equivalent to determining whether or not that value is lower than the r-width (2^r).

Element Update Formula: *antidiagonal* operation type

$$k^* = 2^{r+1} \left\lfloor \frac{k}{2^{r+1}} \right\rfloor + 2^r \left\lfloor \frac{2^{r+1} - (k \bmod 2^{r+1}) - 1}{2^r} \right\rfloor + k \bmod 2^r \tag{C.1}$$

$$v^* : k \bmod 2^{r+1} \begin{cases} < 2^r & \rightarrow v^* = v \cdot m_{10} \\ \geq 2^r & \rightarrow v^* = v \cdot m_{01} \end{cases} \tag{C.2}$$

C.2. Diagonal Gates

Because M is diagonal, regardless of l and r, $\mathbb{I}_l \otimes M \otimes \mathbb{I}_r$ is also a diagonal matrix. Therefore, the label is never affected, so $k^* = k$ (Equation (C.3)). The argument for v^* is the same as for antidiagonal gates. Only the assignment of matrix elements to cases in Equation (C.4) is adjusted for the diagonal structure of M.

Element Update Formula: *diagonal* operation type

$$k^* = k \tag{C.3}$$

$$M := \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \mathbb{I}_2 \otimes M \otimes \mathbb{I}_2 = \begin{pmatrix} a & 0 & b & 0 & & & & \\ 0 & a & 0 & b & & & & \\ c & 0 & d & 0 & & & & \\ 0 & c & 0 & d & & & & \\ & & & & a & 0 & b & 0 \\ & & & & 0 & a & 0 & b \\ & & & & c & 0 & d & 0 \\ & & & & 0 & c & 0 & d \end{pmatrix}$$

$$k_2^* = k_1^* + 2^r$$

$$k_1^* : k \bmod 2^{r+1} \begin{cases} < 2^r & \to k_1^* = 2^{r+1} \left\lfloor \frac{k}{2^{r+1}} \right\rfloor + k \\ \geq 2^r & \to k_1^* = 2^{r+1} \left\lfloor \frac{k}{2^{r+1}} \right\rfloor + (k - 2^r) \end{cases}$$

$$v^* : k \bmod 2^{r+1} \begin{cases} < 2^r & \to m_1^*(k) = m_{00},\ m_2^*(k) = m_{10} \\ \geq 2^r & \to m_1^*(k) = m_{01},\ m_2^*(k) = m_{11} \end{cases}$$

Figure C.2: derivation of the element update formula for dense gates (highlighting boxes for $k = 7$)

$$v^* : k \bmod 2^{r+1} \begin{cases} < 2^r & \to v^* = v \cdot m_{00} \\ \geq 2^r & \to v^* = v \cdot m_{11} \end{cases} \tag{C.4}$$

C.3. Dense Gates

Based on Figure C.2, the terms for the element update formula are derived step-by-step:

- **Element Index**

  1. **r-width**: The first problem to be tackled is the computation of two indices k_1^* and k_2^*, instead of one index. The two non-zero elements in one column are always at the same relative coordinates in adjacent r-blocks. Their distance is hence the r-width, so:
     $k_2^* = k_1^* + 2^r$
     Using this relationship, it is sufficient to derive a formula for k_1^*, which is done in the following steps.
  2. **l-block**: The term and argumentation for the l-block's "starting index" are identical to antidiagonal gates:
     $2^{r+1} \cdot \left\lfloor \frac{k}{2^{r+1}} \right\rfloor$
  3. **r-block and element**: Every r-block consists of 4 diagonal submatrices, every one of these created by computing $m \otimes \mathbb{I}_n$ for an m in M. As a consequence, every column in an l-block has exactly one non-zero element in the "upper" half of that l-block (and one non-zero element in the "lower half"). That makes computing k_1^* trivial, if k falls into the "left" half of the l-block. Since the right half is structurally the same, one can simply "normalize" k into the "left half", if it falls into the "right half". Conveniently, the same operation both does the normalization, and yields the offset with respect to the l-block starting index:
     $k \bmod 2^r$

  Both terms accumulated yield the row index of the first non-zero element in column k, or, in other words, the new element index k_1^*.

- **Element Factor**

  The multiplicative factors follow a very simple pattern. Because r-blocks are strictly adjacent submatrices to an l-block, there are only two options for the multiplicative factors m_1^* and m_2^*, for a

given k:

$$m_1^* = a,\ m_2^* = c$$

or

$$m_1^* = b,\ m_2^* = d$$

The question which option applies, solely depends on whether k falls into the left or into the right half of an l-block. That step of course is not new anymore, it again means comparing $(k \bmod 2^{r+1})$ to 2^r.

The complete element update formula is given below. It additionally contains the term to compute a full result value v_i, whose two summands depend on different input element indices $k^{(1)}$ and $k^{(2)}$ (Equation (3.11)).

Element Update Formula: *dense-gate* operation type

$$k_1^* \left\lfloor \frac{k}{2^{r+1}} \right\rfloor + k \bmod 2^r \tag{C.5}$$

$$k_2^* = k_1^* + 2^r$$

with input vector α, result vector β, $\alpha[m]$ the element in α with $k = m$:

$$v_i^* : k \bmod 2^{r+1} \begin{cases} < 2^r & \rightarrow m_1^*(k) = m_{00},\ m_2^*(k) = m_{10} \\ \geq 2^r & \rightarrow m_1^*(k) = m_{01},\ m_2^*(k) = m_{11} \end{cases} \tag{C.6}$$

with $\quad k_i^{(1)*} = k_i^{(2)*} = k^*$:

$$v_i^* = \beta[k^*] = m_i^*(k^{(1)}) \cdot \alpha[k^{(1)}] + m_i^*(k^{(2)}) \cdot \alpha[k^{(2)}] \tag{C.7}$$

D

Implementation

D.1. Memory Allocation Policy

Algorithm 10 find a free memory entry for a state of q qubits, by iteratively checking and updating a candidate entry against the table of existing memory entries. `mem(q)` returns the memory space occupied by a state vector of q qubits. `abort()` terminates the entire allocation, and sets an error flag. `mem_alloc_table_max_entries` is the static size of the memory allocation table, so the maximum number of quantum states in the system at the same time

```
1:  procedure Allocate(q)
2:      new_address_max = ADDRESS_MAX - mem(q)
3:      if new_entry.address < 0 then
4:          abort()
5:      end if
6:      new_entry = {address = 0, size = mem(q)}
7:      entry_free = false
8:      while entry_free == false do
9:          entry_free = true
10:         for i in mem_alloc_table_max_entries do
11:             if overlap(new_entry, mem_alloc_table_get(i)) then
12:                 new_entry.address =
13:                     mem_alloc_table_get(i).address + 1
14:                 if new_entry.address > new_address_max then
15:                     abort()
16:                 end if
17:                 entry_free = false
18:                 break()
19:             end if
20:         end for
21:     end while
22: end procedure
```

D.2. Databus Conversion

This section provides cycle-by-cycle operation detail on the databus conversion between memory interface (*datastream handshake bus*) and computation engine (*engine data bus*), in the quantum state memory module data back end. It is demonstrated how the incompatible bus signals during the conversion are either supplied externally, or ignored. The examples below use a hypothetical $64\,\mathrm{bit}$ memory interface and $32\,\mathrm{bit}$ vector elements in memory, result in two engine data bus lanes.

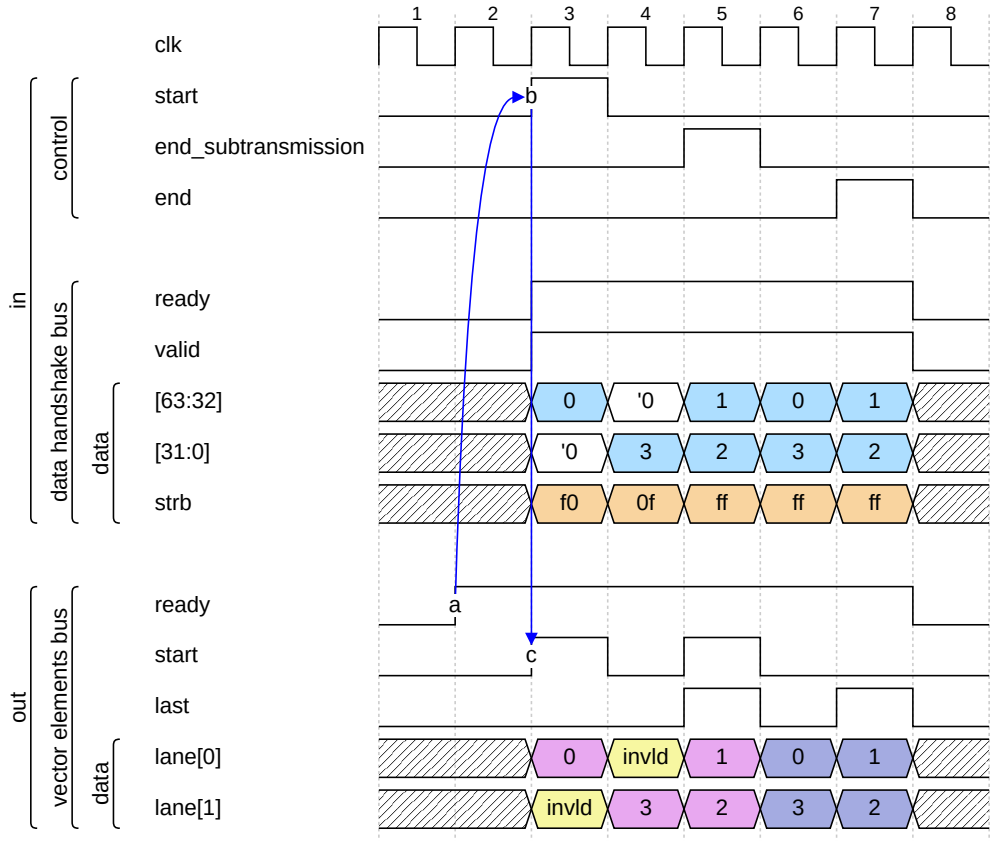


Figure D.1: *Datastream handshake bus* to *engine data bus*. Numbers in data elements indicate labels of the respective state vector elements.

Datastream Handshake Bus to Engine Data Bus

The example in Figure D.1 is a transmission which consists of two subtransmissions. All engine data bus control signals are supplied externally, because the memory interface's datastream handshake bus has no notion of transmissions or subtransmissions.

Engine Data Bus to Datastream Handshake Bus

The example in Figure D.2 is a transmission without subtransmissions. The datastream handshake bus's `valid` signal is determined from a logical `AND` of the *invalid* states of all data lanes.

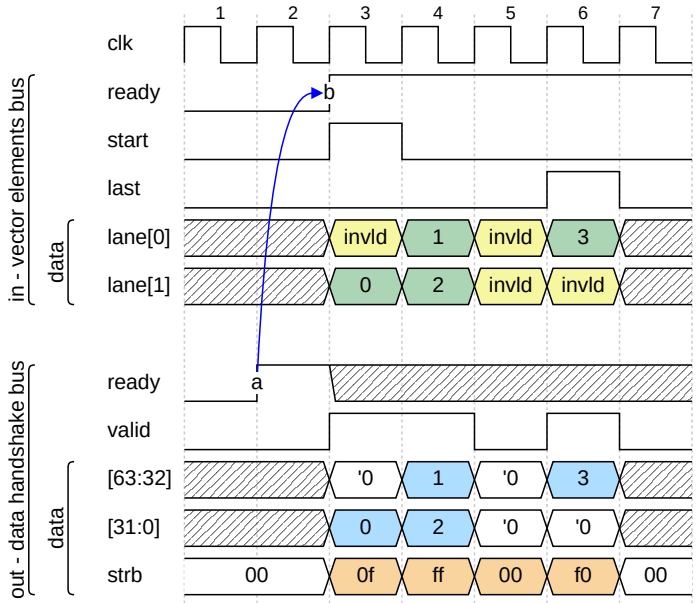


Figure D.2: *engine data bus* to *Datastream handshake bus*. Numbers in data elements indicate labels of the respective state vector elements.