A Hardware/Software Co-designed Partitioning Algorithm of Sparse Matrix Vector Multiplication into Multiple Independent Streams for Parallel Processing

Björn Logi Sigurbergsson

CE-MS-2018-29

Delft University of Technology

Abstract

The trend of computing faster and more efficiently has been a driver for the computing industry since its beginning. However, it is increasingly difficult to continue this trend because current CMOS technology cannot be down-scaled anymore due to physical restrictions. Consequently, to obtain the next major performance improvement, the focus is shifting from a technology-only optimization effort towards a system-level hardware-software co-design optimization strategy. In recent years, the move to heterogeneous computing has gained enormous traction with all the big names such as Intel, IBM, and NVIDIA investing heavily in this approach. This paradigm shift is characterized by traditional general-purpose processors offloading data to *hardware accelerators*, which are capable of exploiting parallelism to a significantly higher degree. An accelerator which has existed for decades but has recently risen to greater prominence is the field-programmable gate array (FPGA). The scientific computing community is also experiencing the need for higher computational power as their problem sizes increase. FPGAs make a promising candidate for their ability to tailor complex algorithms to specialized hardware circuits. A key algorithm to accelerate in this domain is the Sparse Matrix Vector Multiplication (SpMV). There do not exist many HLS (High-Level Synthesis) designs for this kernel, and the one designed using Vivado HLS exhibits significantly lower performance than the state-of-the-art. We argue that the most effective way to achieve speedup is by implementing multiple parallel pipelines so that multiple result values are produced in each cycle. Consequently, we develop an *implementation agnostic* partitioning algorithm for SpMV that splits the problem into independent streams. The HLS kernel performs well as a standalone unit, offering a speedup of up to 150x compared to the ARM co-processor on the ZYNQ system and up to 4.6x to state-of-the-art Vivado HLS-based solutions. Our estimations show that the solution scales with an increasing number of resources.

TU Delft Delft University of Technology

CE Lab Computer Engineering Laboratory

A Hardware/Software Co-Designed Partitioning Algorithm of Sparse Matrix Vector Multiplication into Multiple Independent Streams for Parallel Processing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Björn Logi Sigurbergsson
born in Reykjavík, Iceland

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Hardware/Software Co-Designed Partitioning Algorithm of Sparse Matrix Vector Multiplication into Multiple Independent Streams for Parallel Processing

by Björn Logi Sigurbergsson

Abstract

The trend of computing faster and more efficiently has been a driver for the computing industry since its beginning. However, it is increasingly difficult to continue this trend because current CMOS technology cannot be down-scaled anymore due to physical restrictions. Consequently, to obtain the next major performance improvement, the focus is shifting from a technology-only optimization effort towards a system-level hardware-software co-design optimization strategy. In recent years, the move to heterogeneous computing has gained enormous traction with all the big names such as Intel, IBM, and NVIDIA investing heavily in this approach. This paradigm shift is characterized by traditional general-purpose processors offloading data to *hardware accelerators*, which are capable of exploiting parallelism to a significantly higher degree. An accelerator which has existed for decades but has recently risen to greater prominence is the field-programmable gate array (FPGA). The scientific computing community is also experiencing the need for higher computational power as their problem sizes increase. FPGAs make a promising candidate for their ability to tailor complex algorithms to specialized hardware circuits. A key algorithm to accelerate in this domain is the Sparse Matrix Vector Multiplication (SpMV). There do not exist many HLS (High-Level Synthesis) designs for this kernel, and the one designed using Vivado HLS exhibits significantly lower performance than the state-of-the-art. We argue that the most effective way to achieve speedup is by implementing multiple parallel pipelines so that multiple result values are produced in each cycle. Consequently, we develop an *implementation agnostic* partitioning algorithm for SpMV that splits the problem into independent streams. The HLS kernel performs well as a standalone unit, offering a speedup of up to 150x compared to the ARM co-processor on the ZYNQ system and up to 4.6x to state-of-the-art Vivado HLS-based solutions. Our estimations show that the solution scales with an increasing number of resources.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-29

Committee Members :

Advisor:	Prof. dr. ir. Koen Bertels, CE, TU Delft
Chairperson:	Prof. dr. ir. Koen Bertels, CE, TU Delft
Member:	Dr. ir. Zaid Al-Ars, CE, TU Delft
Member:	Dr. ir. Matthias Möller, DIAM, TU Delft
Member:	Dr. ir. Razvan Nane, CE, TU Delft

Dedicated to my parents and my beautiful girlfriend Sonja

Contents

List of Figures

List of Tables

List of Acronyms

SpMV Sparse Matrix Vector Multiplication

HLS High Level Synthesis

FPGA Field-Programmable Gate Array

MAC Multiply-Accumulate

NNZ Number of Non-Zeros

CU Computational Unit

FLOPS Floating Point Operations per Second

MFLOPS Mega FLOPS

GFLOPS Giga FLOPS

B/s Bytes per Second

PL Programmable Logic

PS Processing System

CG Conjugate Gradient

Bi-CGSTAB Biconjugate Gradient Stabilized Method

DMA Direct Memory Access

HDL Hardware Description Language

I/O Input/Output

SD Card Secure Digital Card

GPU Graphics Processing Unit

CPU Central Processing Unit

RAM Random Access Memory

BRAM Block RAM

DRAM Dynamic RAM

Acknowledgements

The long MSc. journey at TU Delft is finally coming to an end. This has been a very challenging couple of years, full of uncertainties and long working hours but also one of the most rewarding experiences of my life. I want to thank my girlfriend Sonja for her patience and unfailing support. Also thanks to my parents who inspire me constantly with their incredible perseverance.

Many thanks to Razvan for all the patience while this thesis was shaping up, and for being a great guide whose attention to detail kept me on my toes. Thanks to Koen for helping me develop this work from a high-level and for pointing out all the gaps in my explanations. I would also like to thank Tom for all the help and discussions.

I would also like to thank the Skarlet co-founders Saever and Bjarki, as well as Haji and all the hammerboys. My stay here in Delft would not have been the same without the all good times we had!

Björn Logi Sigurbergsson
Delft, The Netherlands
November 5, 2018

1 Introduction

The work presented in this thesis proposes a solution to a computational difficulty in a wide range of application domains. We begin by recounting the rise in prominence of Big Data applications and the rise of heterogeneous computing as a result of Moore's Law prediction approaching fruition. The focus narrows down to scientific computing applications, which currently drive most advanced computer simulations, and their scalability issue. We then define the problem statement and set goals for this thesis to address it. Finally, we present the outline of this thesis and our problem-solving approach by illustrating the relationship between the chapters.

1.1 Context and Motivation

In the age of Big Data, many application domains are seeing an exponential increase in the amount of information to be processed. This is seen in a wide variety of domains such as image processing, DNA sequencing, scientific computing, and machine learning. Furthermore, Big Data Analytics has become a field of its own, which for example enables businesses to do something meaningful with untapped data that collected over the years. The paradigm of computing faster and more efficiently has ruled computing since its conception but is now reaching a saturation point. This was famously predicted by Gordon Moore who made the observation that the computational power cannot continue the trend of doubling every 18 months indefinitely. This is due to physical limits to how small transistors can be made as they approach the size of atoms, thus imposing a bound to the transistor count per integrated circuit. As a result, the effort has shifted towards breaking the workload into independent chunks and processing them in parallel, thereby speeding up computations by a factor of the available computational units (CU).

The move to heterogeneous computing has gained a tremendous traction in recent years. The paradigm shift is characterized by hosts, typically traditional processors, offloading data to powerful heterogeneous systems that act as accelerators by performing in parallel a gigantic number of relatively simple calculations which would take traditional processors unjustifiably long time to compute. The shift is also taking place at an unprecedented rate as is witnessed by the largest names in the computing industry. In 2015, Intel acquired Altera, which along with Xilinx make up the world's largest FPGA (Field-Programmable Gate Array) vendor, for almost $17 billion [4]. NVIDIA and IBM are actively collaborating on the OpenPOWER foundation [5]. Furthermore, large-scale computing is not only limited to wealthy companies after Amazon started the AWS (Amazon web services) initiative. This service enables the public to rent configurable computational power from servers that run GPUs (Graphical Processing Unit), FPGAs and multiple CPUs (Central Processing Unit).

There exist various types of accelerators and the most suitable choice depends on the type of task at hand and the size of the problem. GPUs have become popular in image processing and video gaming as a huge amount of operations on independent pixels are readily mapped to its multiple cores. FPGAs are promising in scientific computing applications as they execute

circuitry tailored to the algorithm, enabling deep task-level pipelines. Finally, clusters consist of multiple machines and are suitable to process data in terms of terabytes and beyond, sometimes in real-time.

The domain of scientific computing is the driver for many applications including simulators which model physical systems both large, such as oil reservoirs [6], and small like cloth simulations [7]. The core of simulators is solving systems of linear equations, which in mathematical terms is to solve for $\vec{x}$ in $\vec{x} = A^{-1}\vec{b}$. The system A is generally very sparse, i.e. the density of non-zero values can be as low as a fraction of a percent, especially if it models a physical environment. As a result, the inverse cannot be directly computed efficiently, if at all. Instead, solver algorithms iteratively *guess* the solution and adjust the result accordingly until it is satisfactory. The computation that dominates the solver algorithm is the multiplication of the matrix with a vector, and the fact that the system is sparse causes a transformation to a new algorithm known as Sparse Matrix Vector Multiplication (SpMV). This algorithm has notoriously low temporal locality of data, making it bound to memory bandwidth. It also has low spatial locality of data so when faced with the task of splitting SpMV for parallel computation, either an expensive merging in the post-processing stage is required or the cache of each accelerator must be as large as the dimension of the problem.

We design a kernel in this thesis which accelerates SpMV on an FPGA, more specifically the ZYNQ-7020 chip from Xilinx integrated on Zedboard. The drawback of developing for FPGAs is that it requires knowledge of hardware design which is a completely separate discipline from computer programming. The designs are described manually in hardware description languages (HDL) and implementing efficient solutions typically takes time in terms of months, even for experienced hardware architects. To alleviate this, FPGA vendors developed HLS (High Level Synthesis) which synthesizes hardware circuits from algorithmic descriptions in imperative languages such as C++. The work presented in this thesis uses HLS to generate the hardware design.

A number of drawbacks of HLS make it a difficult tool to efficiently implement SpMV as noted in [8]. One of which is static scheduling which makes loops with a variable amount of iterations impossible to pipeline with current commercial tools. This is problematic for SpMV because it contains such a loop to process rows with a varying amount of non-zeros. There is ample active research to address this issue [9][10][3] however HLS performance is still likely to remain significantly lower than the state of the art [11][12].

The most efficient solutions exploit parallelism between rows and produce multiple values of the result vector per cycle [11][2]. This is also where current HLS designs fall short due to many intricacies in managing multiple pipelines. In other words, consider the case where a kernel is connected to a high bandwidth bus that delivers multiple rows some cycles, while others only a single row and a part of another row. This dependency for control at run-time arises is the same issue as we described in the previous paragraph. As a result, the kernel can only start processing a single row per cycle, thus the bandwidth can only scale to the size of the row. Currently, there are no SpMV HLS designs that achieve this feat to our knowledge which must be fixed to catch up with their manual counterparts.

Instead of exploring the possibility of maintaining multiple pipelines within a single kernel in HLS, we develop in this thesis a partitioning algorithm which we call *sparstition* for SpMV which splits the problem into disjoint streams. Each stream can be assigned to different CUs on the same fabric, or even different accelerators. Therefore the bandwidth scalability barrier

imposed by HLS can be sidestepped by involving multiple kernels where each one is supplied at most a row each cycle. Naturally, there is extra work involved in the pre-processing stage, but since SpMV appears in iterative algorithms the extra cost is acceptable if it leads to better overall performance. Furthermore, the *sparstition* algorithm is *implementation agnostic* so it works as an addition to any SpMV architecture that exists.

1.2 Problem Statement and Goals

The HLS designs for SpMV are lagging behind those that are described manually in HDLs due to poor bandwidth scalability as row-level parallelism is not exploited. The difficulty arises in managing multiple pipelines as the bandwidth increases to the point where multiple rows are transferred each cycle. This motivates us to ask the following research question.

Can a partitioning algorithm be developed which splits SpMV into multiple independent streams, and show promise in achieving speedup in HLS designs within a reasonable amount of iterations?

To answer this question we set ourselves the following goals:

- Develop a parameterized partitioning algorithm which must only run once in the pre-processing stage.
- Develop a SpMV kernel in HLS which makes use of as much bandwidth as possible, and is capable of processing multiple partitions.
- Co-design the previous two points such that the output from the partitioning algorithm feeds the parameterized kernel, requiring minimal user interference.
- Show that speedup is achievable with a reasonable amount of iterations (i.e. within the number of iterations it takes a decent solver to converge) when all temporal cost has been factored in.

1.3 Thesis outline

The remainder of the thesis is outlined as follows:

- **Chapter 2** delves into what computational modelling is and highlights its importance in various domains. Then it gives details on the SpMV and describes what makes it a formidable algorithm to accelerate. Then we discuss the technology of FPGAs and their role as an accelerator. Finally, we discuss the state of the art designs for the SpMV kernel and how HLS compares with them.
- **Chapter 3** defines the problem with partitioning sparse matrices in order to compute SpMV in parallel. The *sparstition* algorithm is introduced on a conceptual level as a solution. We then introduce the concept of executing *sparstitioned* SpMV in a cluster environment.

- **Chapter 4** describes the design of *sparstition* algorithm which partitions SpMV according to user defined parameters (cache size, number of partitions, etc.). Furthermore, it illustrates the possibility of distributing in a cluster.
- **Chapter 5** describes the architecture of the parameterized HLS kernel. This chapter includes the design both on the block- and code-level. Finally, the task-level pipeline is explained.
- **Chapter 6** implements the designs presented in Chapter 4 and 5, and applies the co-design strategy.
- **Chapter 7** presents the experimental results of the HLS kernel and the *sparstition* algorithm.
- **Chapter 8** summarizes the thesis and highlights the future work.

A high-level schema of the organization of this thesis and the relationship between the chapters is presented in Figure 7.1.

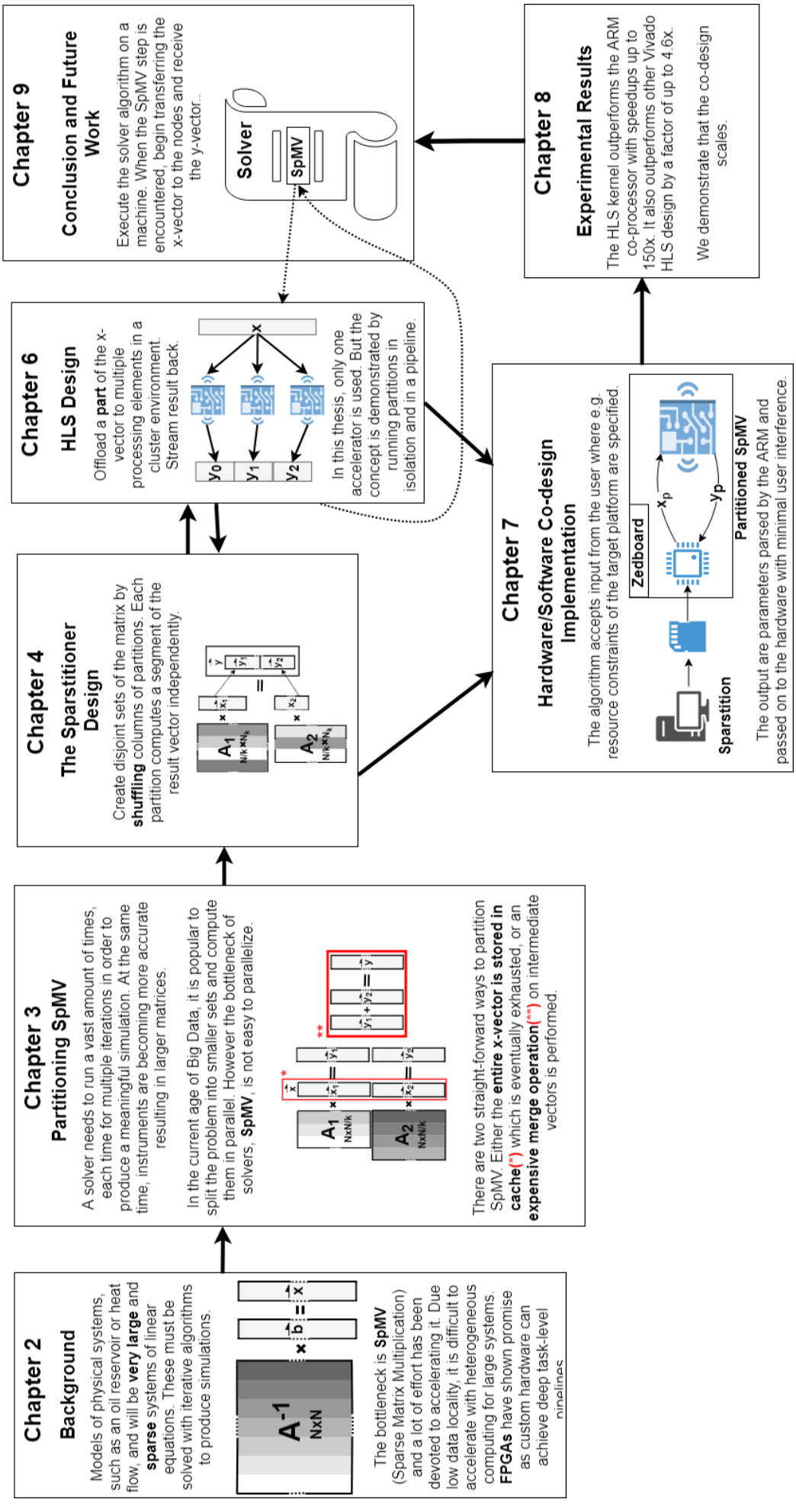


Figure 1.1: The big picture of this work and how it connects to the overall flow.

2 Background

This chapter discusses the motivation for simulating physical systems and how this is achieved with scientific computing. The bottleneck in these simulations is most notably the SpMV and the properties which make it a key algorithm to accelerate are explained. Finally, we discuss the technology of reconfigurable computing, specifically FPGA (Field-Programmable Gate Array), and describe what makes it a strong candidate for acceleration.

2.1 Simulating Physical Systems

2.1.1 Motivation

Simulation replaces physical experimentation by the use of mathematical modelling of a system. The state of the virtual system is defined as variables and the relationships between them, then the model computes the effect on the system when it is subjected to certain conditions. This approach can be very cost effective, for example it would be very expensive to manufacture and test every single variation of an airplane wing to optimize the amount of generated lift. Additionally simulations may speed up the passage of time by predicting the erosion of different materials in water pipes without having to test it for years. Finally simulations help find the effect on systems that are irreversible, for example when finding the optimum drill point in oil reservoirs.

2.1.2 Iterative Solvers

Given a system of linear equations A which is subjected to conditions defined by vector $\vec{b}$, the aim is to compute the resulting vector $\vec{x}$ such that $x = A^{-1}b$. However, in physical systems A is very large and sparse so the direct computation of its inverse is not computationally efficient and possibly not even feasible. Instead the solution is approximated with iterative algorithms that essentially *guess* the solution $\vec{x_{(i)}}$. Adjustments are made based on the result and another, updated guess is attempted. A fitting analogy for this class of algorithms is when a target is shot at with a crooked gun. The location of the bullet is recorded, the aim is adjusted accordingly and another attempt is made. This process is repeated until the result is satisfactory.

A popular iterative solver is the CG (conjugate gradient) method and its many variants, the most efficient and verstaile one arguably being the Bi-CGSTAB (Biconjugate gradient stablized method). The linear system might be characterized by large extremes which translates to large iteration counts in the search for solution. To alleviate this, a *preconditioner* is applied to the system, which is often a costly operation in terms of computation and difficult to optimize, but can pay off by saving a significant number of iterations.

A key property of iterative algorithms is that the sparse matrix does not change. We will exploit this when we consider distributing the workload to multiple computational units. Each

one can keep its own segment of the sparse matrix throughout the duration of the solver without needing to share memory.

The CG methods consist of relatively unintensive tasks such as computing vector norms and dot products, but also the SpMV (Sparse matrix vector multiplication) which dominates the computation time therefore making it the bottleneck. This algorithm is computationally inefficient due to poor temporal and spatial data locality, resulting in a memory-bound kernel which is difficult to parallelize. The application of a preconditioner does indeed become the bottleneck on the micro-level, however on the macro-level it accelerates the algorithm and therefore is not considered an actual bottleneck.

The SpMV is discussed in more detail in the section that follows.

2.2 Sparse Matrix Vector Multiplication (SpMV)

SpMV is a variant of normal matrix-vector multiplication which arrives at the same result. As the name suggests, a sparse matrix contains a very low density of non-zero values, typically less than 1% and even steeping down to just a fraction of a percent. The sparse matrix is compressed in the pre-procesing stage and the algorithm reads the encoding used to avoid redundant calculations. There are many ways to perform the compression, each one requiring a corresponding tweak to the algorithm.

2.2.1 Compression Encoding

In order to avoid performing a huge amount of redundant calculations, pointers to the non-zero values of the sparse matrix are stored. A good measure as to when a matrix is sparse, is when less memory is needed to store these extra arrays than the standard matrix.

The most common (and generic) ones are CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column). There is a wide variety of different compression formats, usually CSR/CSC variants tweaked to exploit sparsity patterns, summarized in a number of other works [13][14][15]. The compression performed by CSR/CSC are depicted in Figures 2.1a and 2.1b respectively.

(a) CSR. (b) CSC.

Figure 2.1: CSR and CSC encodings side-by-side.

Consider a matrix of size $N \times N$ with `NNZ` (Number of Non-Zeros) non-zeros and a vector of size `N` . The `vals` array is of size `NNZ` and contains the value of each non-zero in the order that they appear. There is also an array of pointers to either the row (CSC) or column (CSR) of each value. In CSR it acts as an index to $\vec{x}$ whereas in CSC it is an index to the result vector $\vec{y}$. The *fundamental difference* between the two formats is therefore whether the random access is performed on $\vec{x}$ or $\vec{y}$. The `rows` (CSR) and `cols` (CSC) arrays are of size `N` and convey the information how many elements are in each row/column. SpMV algorithm with CSR encoding is shown in Algorithm 1 and the CSC can be obtained by retrieving the index for the `vals` array instead.

for $i \leftarrow 0$ **to** N **do**
  acc := 0;
  for $j \leftarrow ROW\_PTRS[i]$ **to** $ROW\_PTRS[i+1]$ **do**
    acc += x[COLS[j]] $\times$ VAL[j];
  end
  Y[i] = accum;
end

Algorithm 1: SpMV algorithm with CSR encoding

The `rows` array of CSR is commonly modified to rather store the number of non-zeros of each row. This modification is performed by subtracting two consecutive values and therefore the size of `rows` decreases from $N+1$ to N. From Figure 2.1a, `rows_sizes` is $[2,1,3,1,1...2]$ because the first row has 2 non-zeros, the second row 1 non-zero, etc..

The compression encoding chosen for this thesis is CSR because it is more memory efficient since intermediate results must be stored when using CSC. Therefore SpMV will only be in terms of CSR for the remainder of this thesis.

2.2.2 Properties

The SpMV algorithm has low temporal data locality because each value in the two largest arrays, `vals` and `cols` are used once. The benefit is that these arrays can be streamed, but in turn, makes the application severely memory-bound as the performance depends on how many bytes we can supply the kernel with each cycle.

Furthermore, the algorithm has low spatial data locality as well due to the random access to $\vec{x}$. This property is the source of cache misses if only a part of it can be stored in the cache.

The rows do not have an equal number of non-zeroes so the inner loop has an unpredictable iteration count. This is an issue for tools that perform static scheduling as we will see in the next section when we discuss HLS (High-Level Synthesis).

Furthermore, there is a high number of memory accesses in SpMV relative to the number of operations which we will discuss in the following section.

2.2.3 Computational Intensity

Computational intensity (CI) is a metric for the performance of applications. CI is defined as the number of operations performed on each byte of data fetched from slow memory and the

unit is typically FLOPS[1]/Byte. An application that has high CI performs more computations on every byte fetched and is therefore limited by the availability of computational units. Such an application is *computationally bound* and under these circumstances, the peak performance of the processor is more or less reached. An increase in the computational power of the system results in higher throughput.

On the other hand, an application is *memory-bound* when it performs a low number of operations per byte. If a powerful system has a low supply of bytes, i.e. low bandwidth, then its computational units will be subject to more idle time. In this case, increasing or saturating the bandwidth results in higher throughput.

BLAS (Basic Linear Algebra Subprograms) is a collection of standard matrix-vector operations and is classified into 3 levels according to the CI of the application. The highest level BLAS3 contains for example matrix-matrix multiplication which performs $O(N^3)$ operations to $O(N^2)$ amount of data. The data reuse is high as each row of the first matrix is multiplied with each of the columns of the second.

Next collection is BLAS2 which includes matrix-vector multiplication (MVM) and performs $O(N^2)$ operations to $O(N^2)$ amount of data. There is much less data reuse in MVM as each row of the matrix is now only used once instead of N times. On the other hand, the matrix can be streamed to the kernel during the computation and only the vector needs to be stored in memory.

Lastly BLAS1, which includes vector-vector multiplication, performs $O(N)$ operations on only $O(N)$ amount of data. At most, only the partial sums need to be stored for these routines.

SpMV lies somewhere between BLAS1 and BLAS2, with the amount of data usually leaning more towards that of BLAS1. It is therefore memory bound and also inherits the undesirable trait of BLAS2 to have to store the x-vector in local memory due to random access. The rest of the data can be streamed during the computation of the algorithm.

2.3 FPGA

The process of hardware design has made significant progress since its conception when every transistor was wired manually. As the number of transistors rapidly grew, hardware designers became more dependent on automated tools which enabled them to work at a higher level of abstraction [16]. As the tools developed, the world of hardware design became more accessible. Consequently, specialized sophisticated circuitry can nowadays be generated and synthesized by amateurs and professionals alike with technology such as FPGAs (Field-Programmable Gate Array).

FPGAs contain a vast number of programmable logic blocks which form fundamental computational units such as AND and OR gates, or more complex ones such as combinational functions. The logic blocks are implemented as look-up tables with Flip Flops (FF) as basic memory units and in combination map the input signals to the memorized output signals. These units, which reach millions in large FPGAs, are combined with programmable interconnects which together are capable of forming complex circuitries. This concept is also referred to as programmable logic (PL)

Nowadays complex I/O elements can be integrated into the PL for performance such as DMA (Direct Memory Access) and other memory controllers, digital-to-analog controllers etc.. There

[1] Floating-Point Operations

are two specialized components that are close to the PL and connected with high speed interconnects, one is for performing arithmetic operations, DSP48 blocks, and the other is block RAM (BRAM). Although arithmetic operations can be performed with logic blocks, it is much more efficient to use DSP48 for more complicated procedures such as floating point MAC (Multiply-Accumulate). Likewise, there is significant overhead in forming a lot of memory using only FF, and the BRAMs have the flexibility to coalesce.

2.3.1 ZYNQ

ZYNQ is a small-scale chip developed by Xilinx and is found on some of their development boards, such as Zedboard and PYNQ. ZYNQ consists of a processing system (PS), containing an ARM Cortex-A9 CPU, and programmable logic (PL). On the PS side, all pre/post-processing is executed and data is offloaded to the PL for acceleration, allowing developers to place their own specialized IP cores. The high-level architecture of the chip is depicted in Figure 2.2.

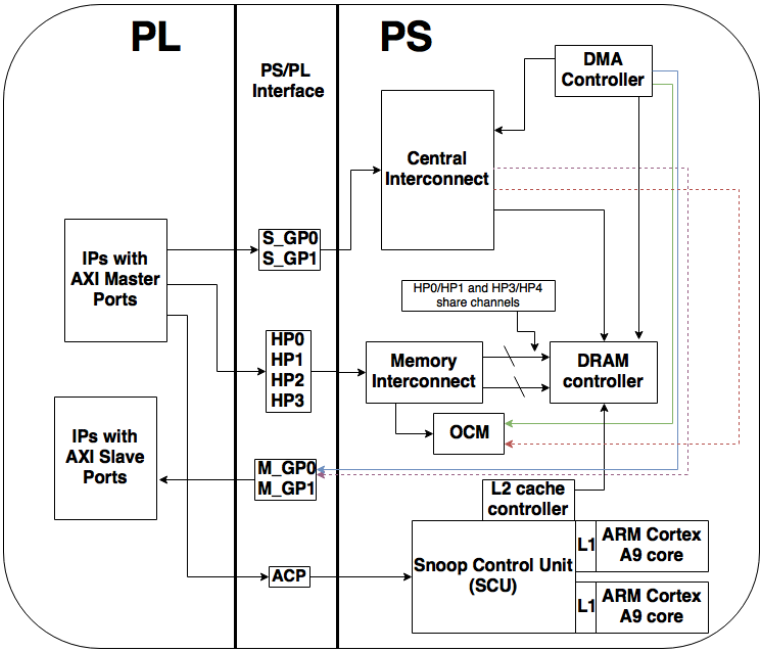


Figure 2.2: Interconnection diagram of the ZYNQ architecture.

There are four different ports on the PS/PL interface. Note that there is only one arrow entering/exiting each interface type for simplification. Each port has its own channel so there should be e.g. four arrows entering/exiting the HP ports box.

- **S_GP**: General Purpose 32-bit wide ports which are slave to the PL. Used to transfer data from PL.
- **M_GP**: Same as S_GP, but are used to transfer data or give instructions to the PL.
- **HP**: High Performance ports, either 32- or 64-bits. Similar to S_GP.

- **ACP**: A single 64-bit port which connects to the L2 cache of the processor, through the Snoop Control Unit (SCU) which is a cache coherency controller for L1 and L2.

The HP ports have a 1KB FIFO to mediate concurrent read/write requests. The theoretical bandwidth of these ports at 150MHz are 1200MB/s and 600MB/s for 64- and 32-bits respectively. Additionally, two HP ports connect to the same channel of the DRAM controller as seen in the diagram. The GP0 port is used by the PS to communicate instructions on which data to access.

A naive way to transfer data to the PL is to write to the memory location (obtained from the block design in Vivado) of the IP block's registers which have been allocated to store the input to the kernel. This approach is useful in testing but is extremely inefficient due to latency in the AXI bus.

The other approach is to use DMA (Direct Memory Access) which provides memory access to AXI-stream peripherals independent of the ARM processor. The DMA is an IP block and can be configured to operate in burst mode to a maximum of 256 data pulses, which blocks the CPU from using the memory bus during the transaction. Alternatively in the transparent mode, data is only transferred when the CPU executes tasks which do not require access to the memory bus. More generally, the DMA converts Memory Mapped data (AXI4) from the DDR to AXI4-Stream (MM2S) and back (S2MM). Essentially AXI4-Stream is for peripherals that *typically focus on a data-centric and data-flow paradigm where the concept of an address is not present or not required* [17].

There are two Data Movers (DM) to consider for the DMA, Scatter-Gather (SG) and Simple. SG mode does not require the memory to be physically contiguous and can transfer larger sizes ($\geq$8MB) at the cost of more latency and hardware resources. This mode also allows the transfer of input sources to multiple output destinations (i.e. buffering). The simple mode on the other hand requires the data to be physically contiguous and only allows the transfer to a single destination.

2.3.2 High Level Synthesis

One of the major barriers for FPGA's to enter the mainstream is the development time. It can take seasoned hardware designers months to develop efficient architectures as there are many nuances which are difficult to account for. Finally, once a design is running as intended in simulation, it may fail to synthesize or not meet timing requirements. This is the impetus for the development of HLS (High Level Synthesis) which is becoming increasingly popular. HLS is one level higher in abstraction from writing hardware designs in HDLs (Hardware Description Language) where the user is not concerned with the design on the register-transfer level (RTL). This has opened the gates for groups of people not familiar with hardware design to synthesize circuits from algorithmic description. They are written in a familiar language such as C++, is translated to the RTL equivalent enabling more focus on the architectural aspect of the problem. There exist a number of advanced HLS tools but in this work Vivado HLS is used.

2.3.2.1 Design Optimizations

The HLS tool may not be able to find the most efficient implementation as it depends on many factors such as the goal of the user. It is also a difficult problem for tools to find certain ex-

ploitable characteristics within the code such as parallelism. The user may direct the tool in the design space according to the goals of the project by the use of **#pragmas**. They are lines of code inserted where the desired effect on the design should take place and may help with anything from optimize loops to the organization of the BRAMs. Below is a brief description of the most widely used pragmas.

A process can be *pipelined* when it consists of instructions or tasks which can be processed in parallel. For example, addition may be divided into three main stages: reading the input, calculation, writing the result. Therefore it is possible to pipeline for example three additions by reading in new input while the second addition is being computed and the third one being written. This concept is illustrated in Figure 2.3, where the non-pipelined instruction execution is demonstrated to take more cycles than if independent operations were overlapped

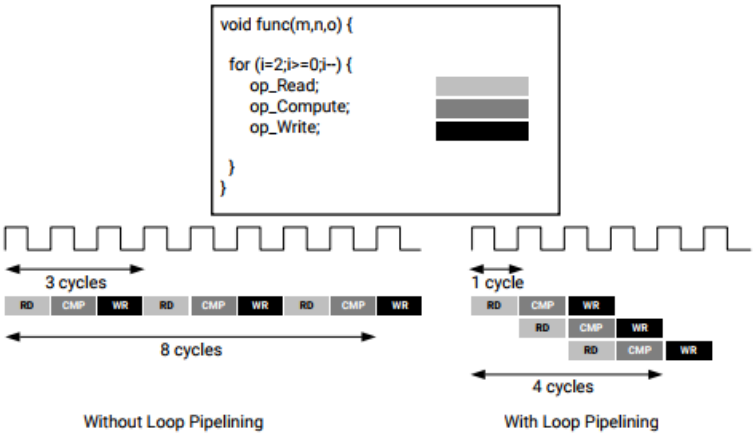


Figure 2.3: Instruction level pipelining [1].

This optimization is applied to loops with the pragma **#pragma HLS PIPELINE II=x**, where II stands for iteration interval. In the example in Figure 2.3, II = 1 because new inputs are accepted every cycle. The pragma is only effective when the loop has a determinate amount of iterations, because the HLS tool must perform scheduling of operations in the circuit.

Pipelining is not limited to instructions but can also be put into effect on the task-level via the use of **#pragma HLS dataflow**. Function calls which follow this pragma are executed in a pipeline given some constraints such as no conditional execution of tasks, no loops with multiple exit conditions etc..

Some loops do not carry dependencies between iterations such as adding together two arrays and writing the result to a third one. These loops can be optimized with a directive known as loop unrolling and is effectuated with the pragma **#pragma HLS unroll factor=x**. The variable x determines whether the the loop is to be fully or partially unrolled. When a loop is fully unrolled, every operation is done in parallel and optimum performance is achieved, however this becomes expensive in terms of resources very quickly.

To achieve parallelism, it is important to consider the number of possible reads every cycle which is determined by the number of ports. For this reason Vivado HLS offers **#pragma HLS array_partition** which gives control to the user how data is organized in memory. Some of the possibilities with this pragma include writing to two BRAMs in blocks or cyclically, or storing each value in a register.

2.3.2.2 Static Scheduling

Current HLS tools perform static scheduling of every operation that is to take place within the generated circuitry. In other word, every arithmetic operation and memory access is given a fixed time slot in synthesis. This does not pose problems for regular algorithms such as standard matrix-vector multiplication. However, SpMV consists of rows that have a varying number of non-zeros which manifests as the inner loop of varying number of iterations in Algorithm 1. The naive solution is to zero-pad each row to the same size, however this issue is currently under active research, as we will see in Section 2.3.3.

2.3.2.3 Development Cycle

The first step when developing HLS kernels is the functional verification, which compiles the code and executes the binary like with normal C code to ensure that that the logic works as intended. Then synthesis is performed which does not always succeed in which case adjustments need to be made to the design. If it does succeed, then statistics such as resource usage and clock period, are available which may trigger the designer to do some further optimizations. At this stage, control diagrams are also available which reveal which tasks are performed in parallel or not. The final step offered by Vivado HLS and also the most time consuming one is the C/RTL co-simulation. This step takes the functional verification in C a step further by also running it using the generated hardware description code. Once this is completed, a waveform diagram is available to the user and if the tests succeed, then the design is functionally sound.

The RTL is exported to an IP (Intellectual Property) and added to the next tool, Vivado. With this tool, the design of the programmable logic takes place which is ultimately used to generate the *bitstream*. The kernel IP is connected to the processing system via interconnects and/or memory controllers such as the DMA. Once the bitstream has been generated, the ultimate resource usage of the design is available along with other statistics such as power usage and whether the design meets the timing requirements.

2.3.3 FPGA Acceleration of SpMV

There exists a great deal of FPGA designs which target the SpMV. As we see from Table 2.1, the performance is directly related to how much bandwidth is available. This observation aligns with our earlier statement that the algorithm is severely memory-bound. Therefore much of the novelty that drives the state of the art forwards is better utilization of bandwidth. A recent novelty [18] proposed exploiting parallelism across rows by introducing multiple pipelines, or channels. An impressive property of this design is that it scales with bandwidth as multiple rows can arrive to the kernel each cycle and start processing without much overhead. It is estimated that even GPUs are outperformed given their bandwidth tier of almost 200GB/s. Other designs have followed suit such as [11][2], all of which are capable of sustaining enormous amount of bandwidth for FPGAs. The efficiency of these designs boils down to producing multiple values for the output vector per cycle.

The state of the art is primarily manual designs except for [2] but it is implemented specifically for Maxeler dataflow engines and will be discussed in more detail in Section 3.4. The only Vivado HLS design [3] for SpMV that we found exhibits performance that is significantly worse than their manual counterparts. Much of the research effort which deals with SpMV-specific

Table 2.1: Comparison of SpMV Works

Works	Device	Benchmark			Performance (GFLOPs)		Bandwidth (GB/s)	
		Name	Dim	Nnz (%)	Single	Double	Single	Double
[19]	Virtex-II Pro XC2VP100	bcsstk06 mcfe af23560	420 765 23560	4.45 4.12 0.087	1.1997 0.9187 0.8786	0.9934 0.7607 0.7275	5.6	7.8
[2]	Stratix V	raefsky1 epb1 scircuit	3242 14734 170998	2.80 0.044 0.0033		3.99 0.69 0.08		38.4 19.2 19.2
[20]	Virtex-5 SX95T	FEM/spheres Wind Tunnel Economics	6.01m 11.52m 1.27m	0.087 0.024 0.0030	17.64		35.74	
[11]	Convey HC-2x	raefsky1 epb1 scircuit				0.55*3.32 0.41*2.50 0.36*2.20		80
[18] (estimated)	Altera Stratix V D5	dw8192 epb1	8192	0.062	2.27 2.45			80
[3] (HLS)	Virtex-7	bcsstm25 poli3 dw8192	15439	0.0065		0.011 0.014 0.024		Simulation

problems is focused on dynamic scheduling to efficiently pipeline the inner loop of Algorithm 1 [9][10] and managing off-chip memory accesses [21][22] when $\vec{x}$ is primarily stored in DRAM. There seems to be little to no investment in developing HLS to receive more bandwidth and producing results from multiple rows through multiple channels each cycle. A part of the reason is that managing multiple pipelines to run in parallel in Vivado HLS is a task with no straightforward solution, if even possible. A naive solution is to assign a port to each pipeline, but the problem with our platform is that each port is 64-bits wide at most. Also to have a port for each pipeline does not scale well.

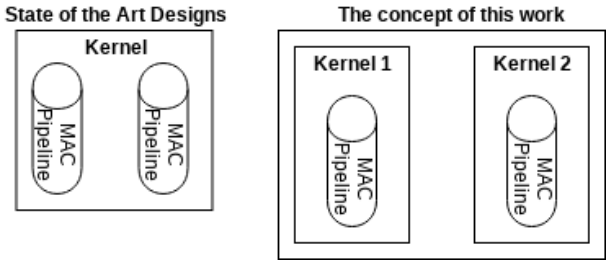


Figure 2.4: High level concept of how HLS design can be made more scalable with bandwidth.

In this work we exploit parallelism across rows thereby activating multiple pipelines in parallel. We do not make any tweaks to the HLS design itself but rather, we partition the problem and distribute the workload to multiple kernels, in theory on the same fabric or even across multiple chips/devices/fabrics etc.. Since this is not a direct improvement to HLS, this concept can be applied to any architecture including the ones in Table 2.1. A high-level illustration of this concept is seen in Figure 2.4 and a more detailed discussion is found in Section 3.2.3.

2.4 Conclusion

In this chapter, we covered all the key concepts behind this thesis. First, we explored the motivation behind simulating physical environments and how scientific computing is the driving force behind them. Then we narrowed the focus to cover SpMV, a recurring bottleneck within that domain. The following section delved into FPGAs and gave details of the ZYNQ architecture which is used to develop the SpMV kernel in this work. Furthermore, we described the main concepts and limitations of HLS. Finally, we discussed the related work on SpMV kernels developed for FPGAs and argued that principle of maintaining multiple parallel pipelines is key to optimum performance.

The chapter which follows dives into a novel partitioning scheme for the SpMV problem, and how that may help boost the performance of not just HLS kernels, but any implementation.

3 Partitioning SpMV

In the previous chapter, we described the context of this work and the technology used. We also introduced SpMV which in this chapter we explain what makes it a challenging algorithm to partition in order to distribute the workload. First, we discuss two naive solutions and point out the difficulties that arise with each one. We then introduce a solution which attempts to alleviate the downfalls of the aforementioned naive approaches which we call *sparstition*. This algorithm basically consists of grouping together adjacent rows and compressing out unused columns. A high-level example with a concrete matrix, namely NORNE, is depicted in Figure 3.1.

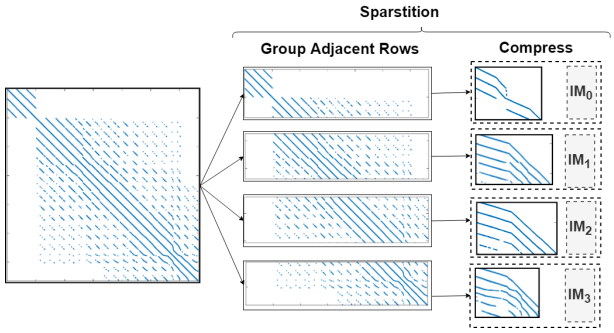


Figure 3.1: The two major steps of the *sparstition* algorithm which will be introduced in tihs chapter.

IM's stand for `Index Maps` and they store the information from the mapping procedure. They will be further explained in Chapter 4. This use case will be a recurrent theme throughout this thesis.

3.1 Naive Partitioning Schemes

This section describes two significant issues with partitioning that arise when naive schemes are applied. First, we describe the merging problem from column-wise partitioning then the issue with row-wise partitioning.

3.1.1 Column-wise Partitioning

Column-wise partitioning cuts $\vec{x}$ at some point k and which corresponds to a vertical cut in the sparse matrix. Consider Figure 3.2, where $\vec{x}$ is cut in half so $k = \frac{N}{2}$ where `N` is the dimension of $\vec{x}$. This produces two smaller vectors $\vec{x_1}$ and $\vec{x_2}$ each of dimension $\frac{N}{2}$ (± 1 depending if `N` is odd) and the matrix is split so that A_1 contains the first half of the columns while A_2 the second half. The two A partitions are then multiplied with their corresponding $\vec{x_p}$ which produces two intermediate result vectors, both of size `N` since each partition still has `N` rows.

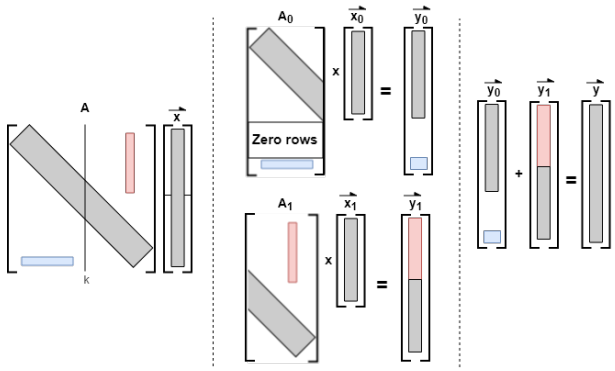


Figure 3.2: Left image shows the partitioned matrix with the chosen cutting point. Middle image is SpMV performed on each partition with smaller $\vec{x_p}$'s. Left image shows the merging necessary to obtain $\vec{y}$ in the post-processing stage.

The benefit of this approach is that the size of the $\vec{x}$ is surgically cut so the $\vec{x_p}$'s can be tailored to fit the available on-chip memory. However, this approach produces partial result vectors $y_1 and y_2$ which must be merged to obtain the final result. This extra step is expensive in terms of hardware and carries communication overhead in a cluster-environment. Alternatively, the reduction may be performed in software by the host, but that is a costly operation especially as the number of partitions grows. Another downfall is that the partitioning produces zero-rows, shown by the empty area of A_0, which need to be efficiently processed in the communication with the HLS kernel presented in Chapter 5.

3.1.2 Row-wise Partitioning

Consider the alternative case where instead of cutting $\vec{x}$ we cut the sparse matrix horizontally. Figure 3.3 shows an example where the halfway point is chosen, so again $k = \frac{N}{2}$. Since each matrix partition A_p contains `N` number of rows, the entire $\vec{x}$ is needed to compute every $\vec{y_p}$.

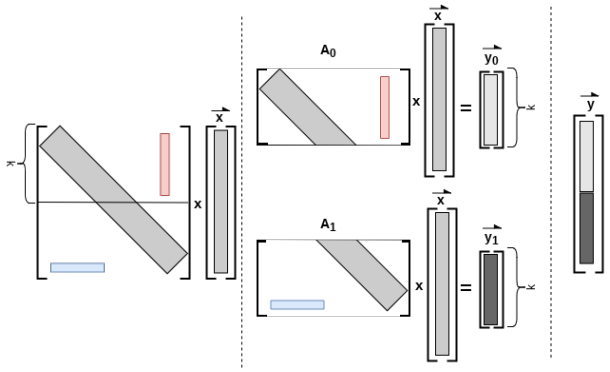


Figure 3.3: Row-wise partitioning computes the SpMV in parallel but requires that the entire $\vec{x}$ is stored in cache.

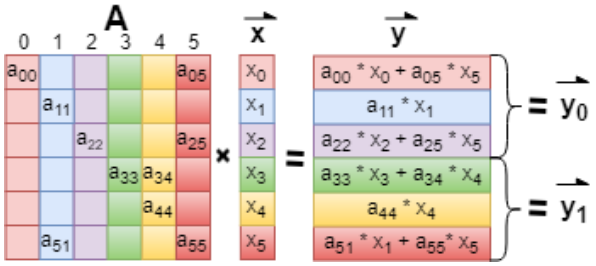
This is a problem if the aim of the algorithm is to fit large vectors in small caches, but the

benefit is that no merging operation is required. We see this in the figure where $\vec{y_0}$ and $\vec{y_1}$ are both of size $k \pm 1$ and each consist of a complete segment of the final result.

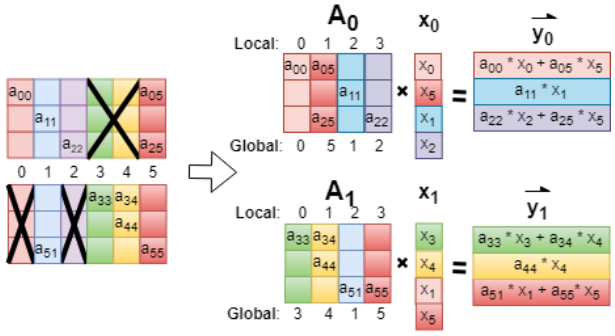
3.2 Sparstitioning

3.2.1 The Concept

A key property of row-wise partitioning is that it does not require the expensive merging step in post-processing. Note that when we split the matrix in this fashion in Figure 3.3, that there is a presence of zero-columns in each partition. This gives rise to the speculation whether *it is efficient to iterate through each non-zero of each partition and shuffle the columns in the order that they appear*. This is the core of the *sparstition* algorithm which we define formally in the following chapter. Consider again a matrix A, vector $\vec{x}$ and k determining the number of rows in each partition, as presented in Figure 3.4a.



(a) The sparse matrix and vector to be multiplied. k marks the point where the cut will take effect. The $\vec{y_0}$ and $\vec{y_1}$ labels refer to Figure 3.4b below.



(b) A after the split contains zero-columns which are compressed out by traversing the non-zeros and assigning them new local columns as they appear.

Figure 3.4: The concept of the partitioning algorithm. The columns of the A_p's are shuffled according to the order in which they appear.

To compute the first value of the result vector, $\vec{y_0}$, we need to calculate $a_{00} \times x_0 + a_{05} \times x_5$, similarly to normal matrix-vector multiplication. Therefore we must keep track of *which columns appear in each partition* as they correspond to which $\vec{x}$ values are required. This is handled with `Index Maps` in the design which are responsible to keep global to local index mappings.

Once k has been determined, in this case, it is the half-way point, notice in the middle diagram that both partitions have two zero columns each, namely 3 and 4, and 0 and 2. This means that $\vec{x}[3]$ and $\vec{x}[4]$ are not needed to compute the upper partition and can be *compressed out*.

The compression consists of traversing the designated rows of each partition, and to *shuffle* the columns in the order they appear in. For example, the first column of A_0 is the same A. The next column, however, is column 5 of A, which is reshuffled to the second column of A_0. The zero columns never occur and therefore do not appear in the resulting $\vec{x}$ partitions, $\vec{x_p}$'s.

A quick sanity check, facilitated by the left image, will reveal that the first half of the result vector $\vec{y} = A \times x$ will be equal to $\vec{y_0} = A_0 \times x_0$ and the second half equal to $\vec{y_1} = A_1 \times x_1$. Therefore, the algorithm has effectively avoided the reduction step.

The characteristics of the algorithm are that each partition may map $\vec{x}$ indices (globa) to different $\vec{x_p}$ (local) indices, depending on the order in which they appear. For example, $\vec{x}[5] \rightarrow \vec{x_0}[1]$ but $\vec{x}[5] \rightarrow \vec{x_1}[4]$. Also, the compression results in smaller $\vec{x_p}$'s, which may enable SpMV to be computed with `N` that normally does not fit in the cache of an accelerator. Also, the order of the summation changes occasionally. For example when computing row 2 of A, $\vec{y}[2] = a_{22} \times \vec{x}[2] + a_{25} \times \vec{x}[5]$ but $\vec{y}_1[2] = a_{25} \times \vec{x}[5] + a_{22} \times \vec{x}[2]$. Finally, the size of $\vec{x_0}$ is rarely equal to the size of $\vec{x_1}$.

The algorithm creates *disjoint* sets of A partitions, each with a new, compressed $\vec{x_p}$. The SpMV can now be computed in parallel, or be made to fit on small on-chip caches. These sets can be thought of independent streams which can be solved in parallel. The algorithm is also *implementation agnostic*, and therefore one half of the matrix could be solved with an FPGA while the other with a GPU.

3.2.2 Exposing Paralellism and Task-level Pipelining

In this section the potential for parallelism and task-level is exposed. The concept is illustrated with a sequence diagram in Figure 3.5 but its implementation belongs to future work. In this thesis, the *sparstition* algorithms partitions sequentially as shown in Figure 6.2. In this example, two unpartitioned reference runs are computed sequentially to demonstrate that the cost of *sparstition* is ideally covered in subsequent iterations.

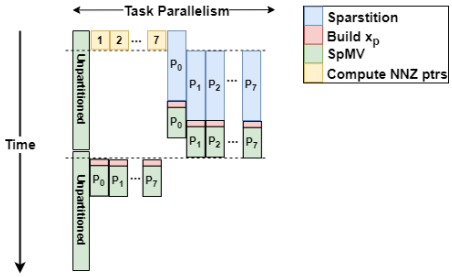


Figure 3.5: Sequence diagram when optimized for parallelism and task-level pipelining on a single machine. This example results in 8 partitions.

We can see in Figure 3.4b that the *sparstition* operation for each partition is independent from each other and thus can be performed in parallel. However in order to unlock this parallelism

we must first slice the matrix data arrays (`values`, `col_ptrs`, and `row_sizes`). The `row_sizes` sizes array corresponds directly to the partitioning point we have chosen, but to slice the other arrays requires us to accumulate each sliced `row_sizes` sub-array. The accumulated results of previous slices is refered to as pre_nnzs_p. This concept is illustrated in Figure 3.6.

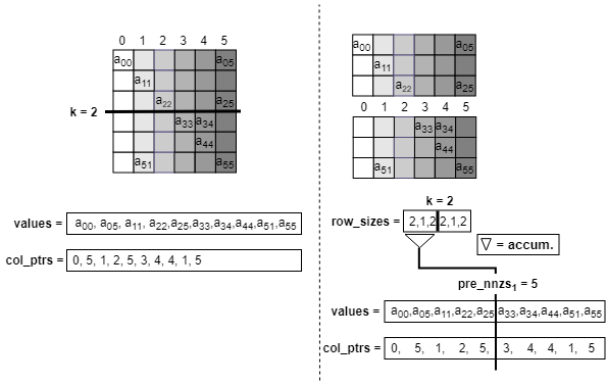


Figure 3.6: Figure that shows slicing of `values` and `col_ptrs` arrays by accumulating row sizes before *sparstition* takes place.

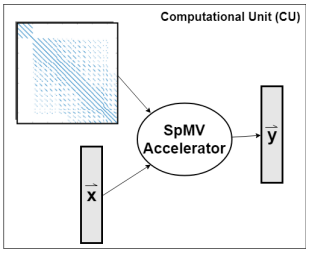
The pointer for the first partition pre_nnzs_0 is trivial because it starts the `values` and `col_ptrs` arrays and is thus always 0. The *sparstition* can therefore start for P_0 as soon as the matrix data has been loaded to memory. The accumulation step can be optimized by using a binary tree, at the cost of more resources. Furthermore, `row_sizes` is an array of integers which is relatively cheap and fast to reduce.

Once the non-zeros for each partition is available, the rest of the *sparstition* tasks can take place, all in parallel. The *sparstition* is a one-time cost in the pre-processing stage, afterwards only building of $\vec{x_p}$'s is required which is relatively cheaper and parallelizable to boot. This is clarified further in the following section.

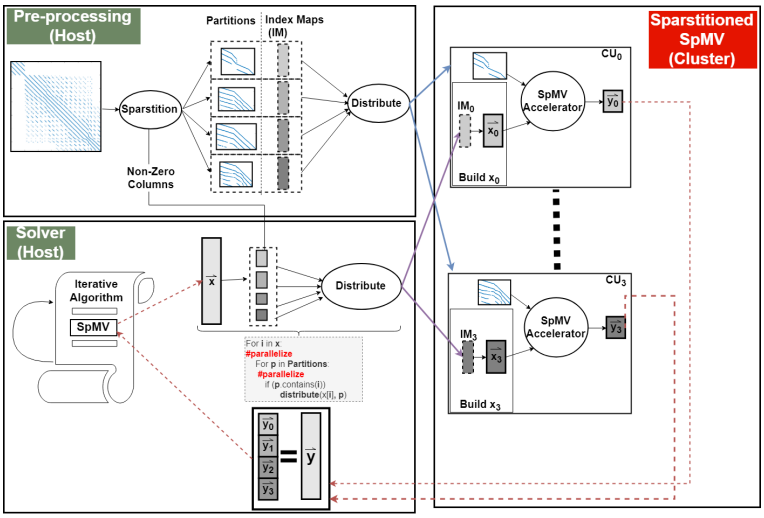
3.2.3 Potential for General Improvement

Although *sparstition* is used to improve HLS performance in this work, an important property which is not exploited is that it can be used in combination with any design. In other words, it is *implementation agnostic*. Once the sparse matrix has been partitioned, all that needs to be done is to *build each* $\vec{x_p}$ and how each SpMV is computed makes no difference. Thus, one partition of the matrix could be computed with GPUs while another with FPGAs, or state-of-the-art kernels could be deployed on multiple machines, each computing in parallel. This concept is illustrated in Figure 3.7.

Figure 3.7a illustrates the standard single accelerator approach where the entire A matrix is computed. The accompanying Figure 3.7b shows the *sparstition* step which groups and compresses adjacent rows and keeps this information in `Index Maps` (IM). IMs are a vital component of the design which will be discussed in the next chapter. The duos (A_p and IM_p) are distributed to whatever disjoint available computational units (CU), different kernels on the same fabric or even machines in a cluster, where they are stored through the duration of the solver. Recall that an iterative algorithm does not update the input sparse matrix.



(a) A single CU from a high perspective.



(b) Multiple CUs which may consist of different types of accelerators.

Figure 3.7: Proposal how *sparstition* may accelerate the state-of-the-art by encasing their implementations in computational units (CU) and multiplying them. It is possible in theory to only distribute needful $\vec{x}$ values to each CU to save bandwidth.

Each time the solver needs to compute SpMV, $\vec{x}$ is distributed, or broadcast, to each machine where $\vec{x_p}$ is built. There is potentially room for optimization in this step, as the building of $\vec{x_p}$ is fully parallelizable. Ideally, only the required values are distributed to each accelerator to save bandwidth and this is possible with `empty_regions` (ER). ERs are produced when compressing `Index Maps` and contain information on where the non-zero columns are located. More information on ERs is found in Section 6.2.2.

There are indeed many new tasks introduced involved in transferring data to and from each CU in a cluster environment. We aim to address these new challenges by exposing where the parallelism lies in the following section.

3.3 SpMV with Big Data Frameworks

There is no upper limit on how large a sparse matrix may grow, especially now in the era of Big Data. Solutions need to be found to linear systems which are growing in size due to, for example, higher resolution instruments. The computation of such a large-scale problem is a daunting task for a single machine as its limit will always eventually be surpassed. This motivates the

partitioning of the problem into disjoint streams as discussed in previous section, and assign each stream to a dedicated node in a cluster. The implementation of such a cluster solution is out of the scope of this thesis, but a potential schema will be discussed in this chapter for future work.

3.3.1 Cluster Schema

There are two types of nodes in the schema under consideration, the driver node and the worker nodes. Usually a master node is included which manages the shared storage but for this explanation it has been left out. The entire process in a cluster environment is illustrated in Figure 3.8.

The initialization is shown separately but in iterative algorithms it is reasonable to assume that the time it takes will be masked by the speedup gained from the partitioned problem. It is divided into 4 tasks, the first one (1.1) loads the data into program memory, task (1.2) then uses the `rows` array to compute the pointers labeled as ptr_i but in the implementation the identifier used is *pre_nnzs*. These pointers are computed by adding up the sizes of the rows belonging to the partition. This is necessary in order to find out where a partition begins in the values and column pointer arrays. Once the number of non-zeros for each partition has been computed, the array can be sliced and *sparstition* performed in parallel. Next step (1.3) is then to transfer the sliced matrix data arrays to its designated node where *sparstition* (1.4) is performed. The global column pointers do not need to be stored intermediately, but if the implementation requires it, they can be adjusted in-place. The output `Index Maps` are cached on each node.

Once initialization is completed the computation of the solver can start, which may be accelerated even though it is not included in the schema. When SpMV is encountered, the vector is streamed/broadcast to the worker nodes (2.2). The simplest solution is to transfer the entire vector but optimally only the desired values. This may be possible in some Big Data frameworks by making use of the `empty_regions` introduced into Section 6.2.2 since the vector values are needed in sequential order, but that needs to be researched further. Finally each node builds its x_p and computes SpMV which it streams back to the driver that continues its computation. Recall that the *sparstition* algorithm is *implementation agnostic* so each node may compute its part of the result using different architectures or accelerators. Optimally, the driver pipelines the SpMV computation with the rest of the solver algorithm.

3.3.2 Sequence Analysis of the Cluster Tasks

To understand which tasks can be run in parallel and where the pipeline potential lies, consider Figure 3.9 which is an extension of Figure 3.5 for cluster.

The main difference is that now reading of the matrix data and the transfer tasks to and from worker nodes is accounted for. The first set of *sparstition* takes place shortly after the transfer of matrix data has started, between Stage 1 and 2, as it is assumed that these tasks can be pipelined in Big Data frameworks. As before, the first node can begin *sparstition* first due to not needing pointers from task (1.2). In Stage 3 the first Worker Node is ready to accept jobs and the transfer of $\vec{x}$ begins. In Stage 4 the rest of the Worker Nodes are ready and in Stage 5 the first iteration is complete enabling the next iteration to begin.

Deploying the solution to a cluster is an expensive and time-consuming task, especially if all the parallelism is to be exploited. Before that step is taken, the concept is first tested

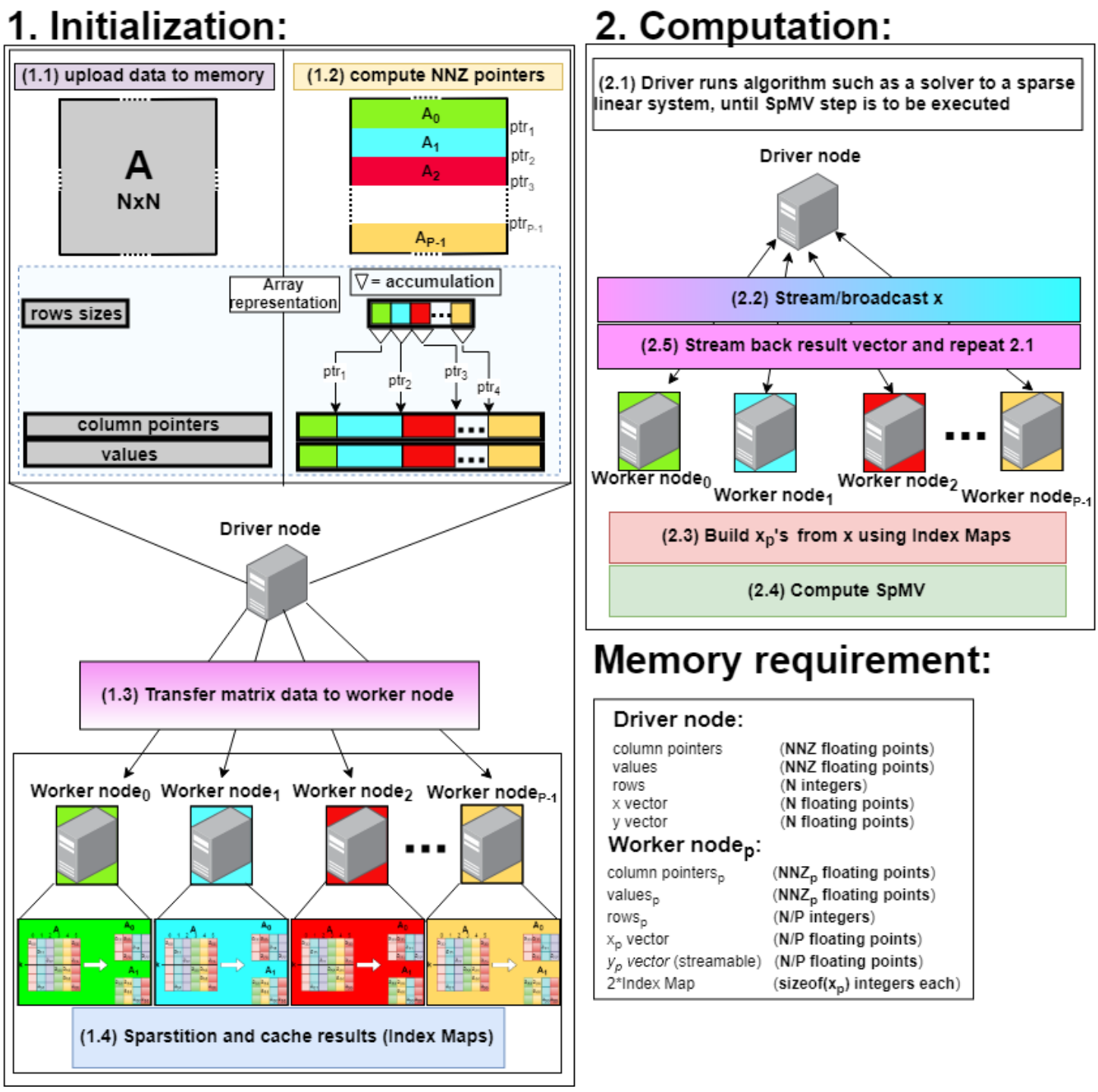


Figure 3.8: Schema of cluster execution of a solver with SpMV distributed to multiple nodes. Colors of tasks correspond to Figure 3.9 and the *sparstition* images are found in more detail in Figure 3.4.

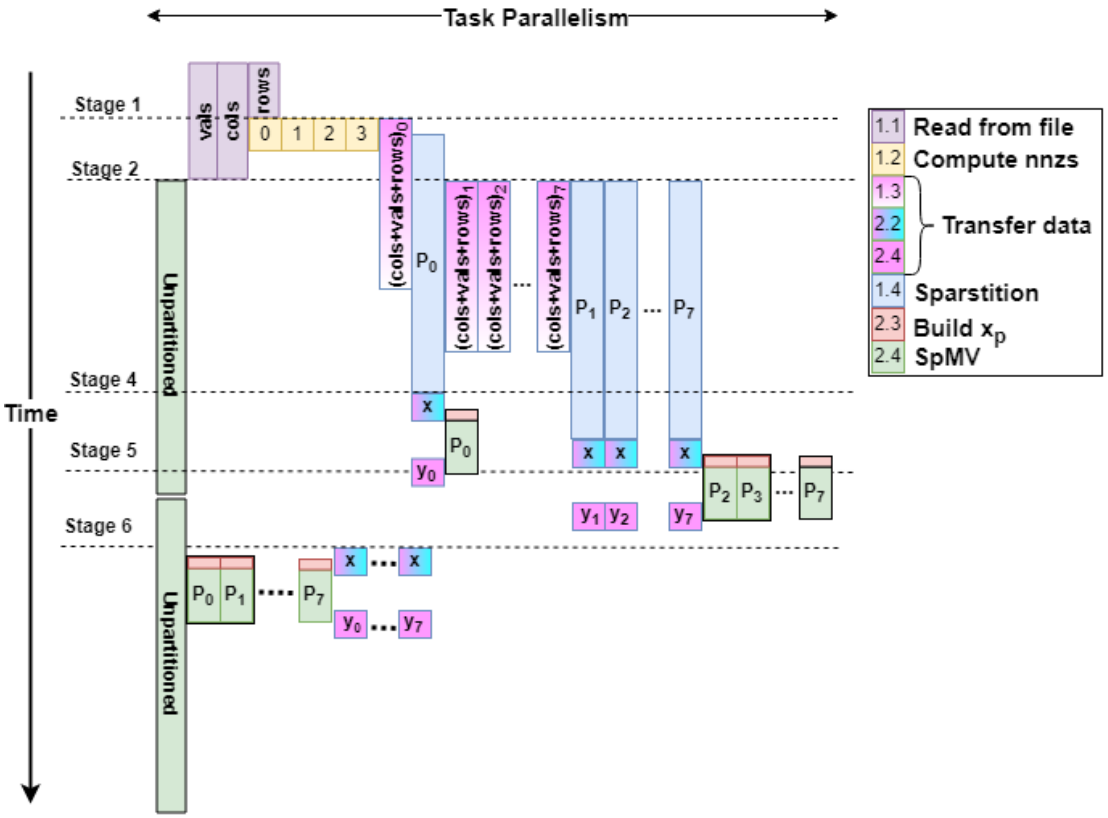


Figure 3.9: Sequence diagram of tasks running in a cluster environment. Labels in legend refer to Figure 3.8.

on a Zedboard to verify that there is a speed-up when pre-processing time is factored in. We execute each partition *in isolation* and take the longest partition to mean the total run time of the application. Then we compute the total execution time from the start of *sparstition* until the result vector has been streamed back to the DRAM. None of the exposed parallelism is exploited in the *sparstition* algorithm so there is much room for improvement in the future.

3.4 Related Works

We discuss in this section firstly CASK which is a promising solutions which achieves multiple pipelines within a single kernel and we compare it with our solution. Then we recount some of the more interesting partitioning solutions that have currently been published.

3.4.1 CASK

We stretched the importance of multiple parallel pipelines in Section 2.3.3. Additionally, we claimed that HLS will continue to lag behind solutions that are designed manually in HDLs (Hardware Description Language) until this feat is achieved.

CASK (Custom Architectures for Sparse Kernels) [2] is very promising but it is implemented

on Maxeler dataflow engines and is not shown to be realizable with standard HLS tools. The pre-processing stage shares similarities with *sparstition* as it performs partitioning by first grouping adjacent rows together but the fundamental difference lies in the columns processing. Instead of shuffling columns like in the *sparstition* algorithm, independent sets of columns are formed with an operation named *blocking*. This step guarantees linear access to the $\vec{x}$ and therefore shares similarities with CSC (Compressed Sparse Column), although the matrix is indeed encoded in CSR. The intermediate results of every row in the partition must therefore be stored to be reduced in the following stage with the so-called *inter-block reduction unit*. Thus *blocking* combines elements of CSC in a clever way which has the advantage of linear access of $\vec{x}$, but requires storing and reducing intermediate values. This issue is avoided by column shuffling, but each $\vec{x_p}$ must be built before computation starts. Lastly, CASK is not *implementation agnostic* as its efficiency is only available to designs that access $\vec{x}$ linearly. On the other hand, *sparstition* enables multiple dataflow engines running CASK to solve a large-scale SpMV problem in parallel. Table 3.1 summarizes the above comparison.

Table 3.1: Comparison between CASK [2] and *sparstition*.

CASK	***sparstition***
Groups adjacent rows	
Supports Multiple Streams	
CSC-esque processing of $\vec{x}$	Shuffling of columns
Merging of intermediate results necessary	Requires building of $\vec{x_p}$ each iteration
Implementations processing $\vec{x}$ sequentially	Implementation agnostic

3.4.2 Partitioning Solutions

One of the most popular ways to perform partitioning in SpMV is to transform the matrix A into a graph G_A. Each vertex represents a row and an edge is drawn between them if they have a column index in common. The goal is to find a partition which minimizes the number of edge cuts, because when connected vertices belong to two different partitions, the corresponding $\vec{x}$-value must be transferred twice which which means communication time is higher.

In [23], the authors develop a multi-level partitioning scheme which basically coarsens the graph so that it shrinks to a few hundred vertices and bisection is computed. Then the bisected, smaller graph is uncoarsened, with each step further refining the partition, and eventually the result is projected on the original graph. The partitioning time is relatively long compared with expected run-time of SpMV on the original matrix. For example, to perform a 256-way partitioning on `ADD32`, a matrix of 23884 non-zeros, just over a second is required which is still a significant improvement over the compared works in the paper. The benefit is, however, that the resulting partitions have similar number of non-zeros. This case was not considered for this thesis since the HLS kernel treats every row as if they were the same size. As a result, partitions with the same amount of rows have effectively equal amount of non-zeros. This is discussed in more detail in Section 5.3.

This partitioning scheme is used within AmgX, a library for NVIDIA's CUDA framework which accelerates methods in finding a solution to sparse linear systems [24]. Once the matrix

has been partitioned, the diagonal blocks are shifted to the left by changing the column indices, resulting in the blocks to be stacked in a vertical line. This is similar to the *sparstition* algorithm, however the matrix has been partitioned into blocks, so column-wise and row-wise, in [24] which gives rise to a new classification of rows and vector elements. Rows are classified into interior, boundary and halo rows where interior rows fit entirely within the diagonal block, boundary rows have columns outside the block as well and halo rows are entirely without. In a similar fashion, $\vec{x}$ is classified into local (boundary and interior) and halo elements, with the local elements appearing first in the $\vec{x}$ followed by the halo values appended to the end.

The SpMV takes part in two stages. The first one computes the interior rows and overlaps it with the communication of the halo elements of $\vec{x}$. In the second stage, the remaining computation for the boundary elements takes place. The merging of intermediate result vectors is not required in this library similar to *sparstition*.

3.5 Conclusion

In this chapter we began describing two naive partitioning schemes and pointed out the problems with each one. We then described our leverage of them with the *sparstition* algorithm in terms of avoiding expensive post-processing and exceeding caches smaller than `N`, the dimension of the problem. This is summarized in Table 3.2.

Table 3.2: Comparison of two standard partitioning schemes with the work of this thesis.

Partitioning Scheme	**Size of Cache (S_C) Required for Each Partition**	**Merging Necessary**
Column-wise	$\frac{N}{N_P}$	✓
Row-wise	N	✗
sparstition	$\frac{N}{N_P} \leq S_C \leq N$	✗

We then exposed parallelism in the *sparstition* algorithm to be exploited in future work, but essentially each partition can be *sparstitioned* in parallel once the matrix arrays have been sliced. We finally transferred our findings to a hypothetical cluster environment where an attempt was made to identify tasks that must take place on a conceptual level. Lastly, we finished the chapter with related work. Special attention was given to CASK which generates multiple streams within a single kernel, and pointed out the differences to *sparstition*. We then briefly explained the state of the art partitioning schemes which aim to balance out the number of non-zeros with advanced graph partitioning techniques.

The following chapter outlines the design of the *sparstition* concept introduced in this chapter. The chapter describes

4 The Sparstitioner Design

We introduced the concept behind *sparstition* in the previous chapter and exposed the parallelism which can be exploited for future implementation. In this chapter, we look at the solution more concretely and present the design of the `Sparstitioner`, the framework which encases the algorithm. The framework consists mainly of the *sparstition* algorithm and the management of the partitions. The main feature is that the framework is *parameterizable*, so that, for example, the partitions can be made fit in the cache of the target accelerator. Finally, we look into load-balancing and we explain that there are two ways to perform it.

4.1 Overview

4.1.1 Definitions

We define the following terms for this thesis that arise in this chapter.

- *sparstition* is the novelty of this thesis and the term is derived from the combination of the words *sparse* and *partition*. It is an algorithmic step which consists of **grouping together adjacent rows** and **compressing out zero-columns** which appear as a result of the first step. The compression is performed by **column-shuffling**. The resulting compressed partitions are *sparstitioned*.
- `Index Map` is a data structure which exists for each *sparstition* and holds mappings from global column pointers (pre-*sparstition*) to local column pointers.
- `Sparstitioner` is the framework which consists of two functions. Namely performing *sparstition* and processing `Index Maps`. It is also capable of meta-functions, such as writing results to files.

4.1.2 Design Goal and Requirements

The goal of *sparstition* is to split the SpMV computations into **disjoint** sets with no expensive operations in the post-processing stage. This enables parallel computation between computational units (CU) that do not necessarily share memory, and with each one computing a set of rows. There are two main reasons to perform partitioning, one is to distribute the workload to multiple CUs for speedup, and the other to split a large $\vec{x}$ so that each segment fits in the available cache of a separate accelerator. In the latter case, a single *sparstition* may not suffice which will require *sparstition* to take place *recursively* until constraints are met.

To summarize, the goal of the *sparstition* algorithm is the following:

- Split the problem into disjoint streams, taking into account the problem characteristics (e.g., input array sizes) and hardware constraints.

- Recursively split the problem until it meets the constraints.
- Functionally verify the correctness of the partitioning.
- Apply safeguards to handle cases when an ineligible matrix is to be processed.

4.1.3 Algorithm Parameters

The parameters are as follows:

- Floating point precision (single/double).
- Maximum size of each $\vec{x_p}$.
- Number of partitions (N_P)

  Enable `GREEDY` (see Section 4.2.1).
- An identifier of the target matrix.
- Write results to external files enabled.

  Output location of results.

  Zero-pad to the bandwidth (see Section 5.3).
- Method of processing `Index Maps` (see Section 6.1.2.

4.2 Algorithm Design

Given a sparse matrix A that we want to partition, the first thing to consider is where is the optimum point so that the result is as balanced as possible. This is important especially in cluster environments where the total execution time of the application is determined by the machine of the longest duration. However, for this work it will not make much difference as long as the number of rows is kept equal for reasons clarified in Chapter 5. Therefore each time *sparstition* is performed, the numbers of rows is split evenly.

Let us expand the concept presented in the previous chapter in Figure 3.4 with a more concrete example. Figure 4.1 splits the matrix A in half as in the previous example.

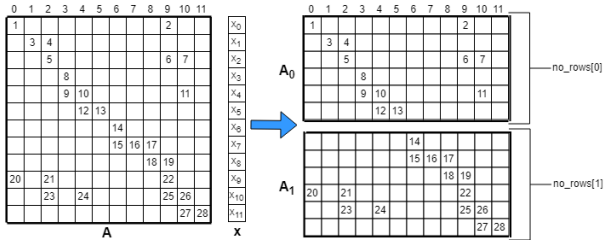
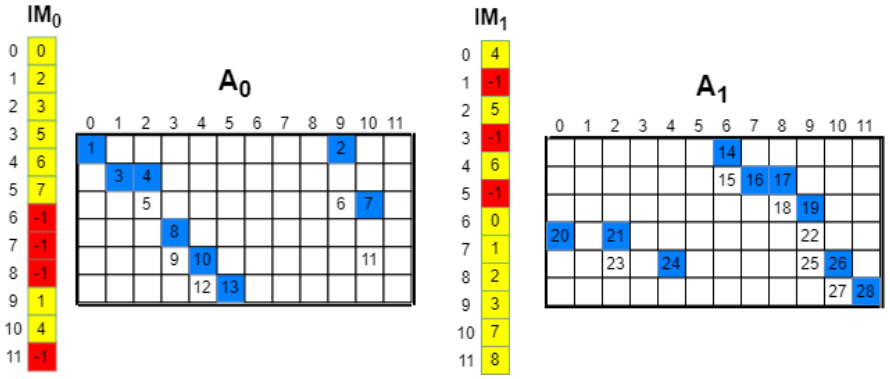


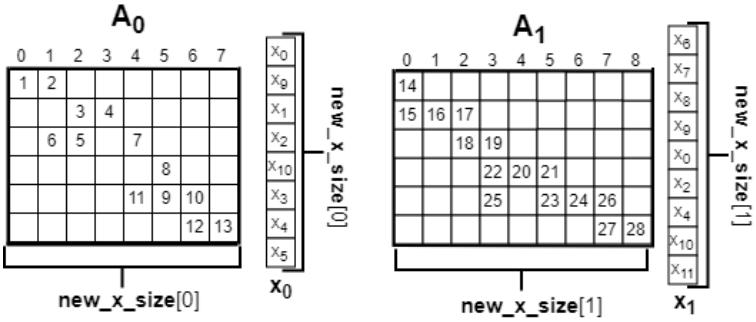
Figure 4.1: Splitting up A to A_0 and A_1, each with *no_rows* amount of rows.

With the new A_p partitions, we notice again the presence of zero columns, for example A_0 has zero-columns 6-8 and 11. If we perform a compression by creating a mapping without them,

then we have effectively created *smaller $\vec{x}$ segments*, or $\vec{x_p}$'s. Let us define an `Index Map` from the global $\vec{x}$ indices to local $\vec{x_p}$ indices, *for each partition*. Each map is made by iterating through the columns pointers of the non-zero values and sequentially assigning them new pointers as **new columns occur**, as seen in Figure 4.2a. In other words, the matrix data is **split and compressed** which results in $\vec{x}$ being transformed into smaller $\vec{x_p}$'s as seen in Figure 4.2b. This combination of tasks is the *sparstition* algorithm.



(a) Constructing `Index Maps` which map $\vec{x}$ indices to x_p ones, for every partition. The matrix is parsed row-by-row, from left to right, and when a "new" column is encountered (shown as blue squares), a new mapping is created. Yellow squares are initialized and red ones are uninitialized thus denoting a zero column.



(b) The compressed partitions. The zero-columns have been removed

Figure 4.2: The partition compression process

There exist three different implementations to keep track of which mappings have been made which are discussed in detail in Section 6.1.2. For now, assume that each `Index Map` has every value initialized to -1 to indicate empty mapping. Once the initialization is complete, the *rows* array is sliced into two parts and the column pointers which belong to those rows are parsed. When a new column pointer is encountered, a new mapping is made in the `Index Map` and *the original column pointer is adjusted*. So in the example of IM_1 in Figure 4.2a, the `col_ptrs` array is [6,6,7,8,...]. The counter of the new $\vec{x_1}$ starts at 0, which is assigned the first column encountered, namely column 6. Column 6 appears again but since `Index Map` [6] $\neq -1$, a new mapping is not made but only the column pointer is updated. Then column 7 is parsed which creates a new mapping with the subsequent value 1. Finally, the `col_ptrs` array becomes [0,0,1,2,2,3,4...]. Note that the `Index Maps` only need to be constructed once, and can be used

to split any $\vec{x}$. This is an important property of *sparstition* as it can be regarded as one-off cost. We will make extensive use of this property in the implementation.

At this point a couple of metrics are at our disposal. The obvious one is the `no_rows`, which is under complete control in this partitioning scheme. The second metric is `nnzs` which is the number of non-zeros in each partition, in the example above *nnzs*[0] is 13 and *nnzs*[1] is 14. These metrics are obviously not affected by *sparstition* and are used to slice `col_ptrs` and `values` array back in Figure 3.6. Finally there is the number of non-zero columns, in other words the size of the resulting $\vec{x_p}$'s or `new_x_size`. As can be seen, this metric is determined by the nature of the rows within the partition. If a resulting $\vec{x_p}$ still does not meet the memory constraints, i.e. there are too many non-zero columns in the partition, then another *sparstition* needs to be performed.

4.2.1 Multiple Partitioning

In order to design a scalable solution, the *sparstition* algorithm must be able to partition the SpMV multiple times resulting in a corresponding number of partitions (N_P). Two cases give rise to this need, namely:

1. The problem must fit on a single accelerator with limited amount of cache.
2. There are multiple accelerator and each one should process at least one partition.

To accommodate the first case, the parameter `CACHE_SIZE` is exposed to the user which, along with the floating point precision, determines the maximum size of each $\vec{x_p}$. The benefit of performing bi-partitioning is that there is only one degree of freedom. If more degrees of freedoms were added, such as in tri-partitioning and beyond, then the complexity increases polynomially. This approach is practiced commonly [23].

The parameter `NO_PARTITIONS` is exposed for the second case and does not necessarily have to equal the number of accelerators. The two parameters may work together in case `NO_PARTITIONS` for the given `CACHE_SIZE` constraint.

4.2.1.1 Meeting the CACHE_SIZE constraint

The optimal solution to the first case is found *recursively* as seen in Figure 4.3. This is also defined as *recursive bi-partite partitioning*. The algorithm starts with a single *sparstition* and since the example matrix has halves of different densities, $\vec{x_1}$, which corresponds with the lower half, consequently does not meet the constraints. This triggers *sparstition* operations until each $\vec{x_p}$ is small enough for the available cache.

The recursive partitioning is formulated with a **binary tree** where the root represents $\vec{x}$, and the leaves $\vec{x_p}$'s. The node is a representation of which rows have been assigned to which partition. The red nodes convey that partitioning has been performed on that segment of the matrix, and all of its associated memory can be freed. The labelling of the white nodes indicates where the rows have been assigned. So in Figure 4.3 to the right, the first half of the rows are to be found in the node labelled 1. The rows that start where node 1 ends are to be found in 2, and so forth.

When *sparstition* is performed on a partition, such as A_0 in Figure 4.2b, notice that *new* zero-columns are produced, namely columns 5-7. Thus it is possible to construct the new `Index`

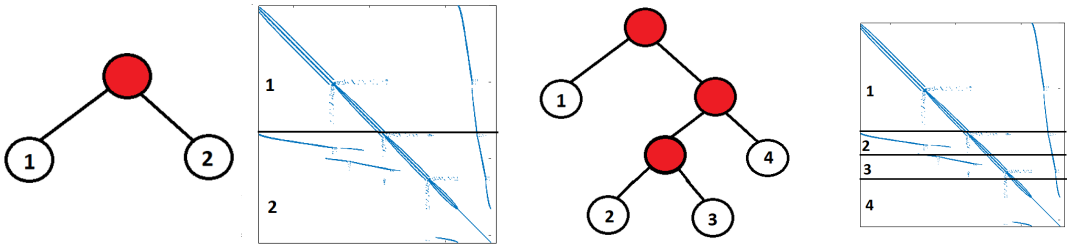


Figure 4.3: The result after 1 (left) and 3 (right) *sparstition* operations after recursively searching for a solution to `CACHE_SIZE` constraint. Note that the leaves are kept in sequential order from left to right.

`Maps` using either `col_ptrs` from A or A_0. Recall that each *sparstition* operation produces an `Index Map` to link the previous partition with the new one, and is used build $\vec{x_p}$'s from $\vec{x}$. The latter is preferable as otherwise all `Index Maps` from the leaf to the root are needed in order to build $\vec{x_p}$, costing much more time and memory.

4.2.1.2 Meeting the NO_PARTITIONS constraint

The second case results in at least N_P partitions, defined by the user via the parameter. It is unnecessary to execute the relatively expensive *sparstition* recursively as was done in the previous case, especially if performance is a concern. Rather, the rows array is sliced straight away into `NO_PARTITIONS` partitions *recursively*, which results in N_P slices of the row arrays each of roughly equal length. The limitation is that N_P must be a power of 2, so perhaps this will need to be addressed in the future.

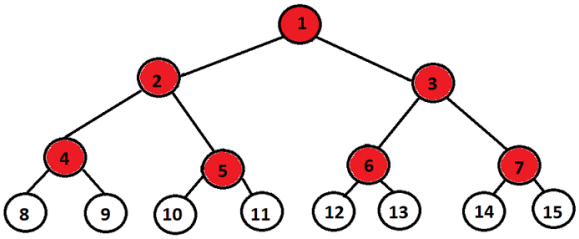


Figure 4.4: The tree resulting from meeting the `NO_PARTITIONS` constraint.

The tree shown in Figure 4.4 is the result. Notice that it is more balanced than the tree in Figure 4.3. The *sparstition* algorithm is only performed on nodes 4-7 and the white nodes are the final partitions. The number of rows is equal for every white node but the number of non-zeros and the size of $\vec{x_p}$ depends on the sparsity pattern. We will argue in the following section that keeping the number of rows equal has the largest effect to balance the partitions.

4.2.1.3 Interplay of CACHE_SIZE and NO_PARTITIONS

The two parameters are not mutually exclusive. The user might request N_P that is too small given the input matrix and the configured `CACHE_SIZE`. When this is the case there are two options

available, to recursively find a solution like in the first case, or to maintain the load-balanced structure of the tree. The former option is potentially cheaper as it requires less *sparstition* operations but the latter keeps the number of rows between partitions equal. This is up to the user to decide so a parameter is exposed under the name `GREEDY`. If `GREEDY` is activated, then a solution is found recursively.

4.3 Load Balancing

4.3.1 Modelling Execution Times with Weights

An important observation before developing a model to predict the execution times of the partitions is that **every row is effectively the same size** in the HLS architecture due to zero padding. However, before computation can begin $\vec{x_p}$ must be transferred to the kernel as we will see in Section 5.3. The two most important components which determine the execution time are therefore the *size of* $\vec{x_p}$ and *the number of rows*.

Obviously each row is more expensive to transfer than a single $\vec{x_p}$-value. Once $\vec{x_p}$ ends and the computation of SpMV begins, it takes one cycle to fetch the size of the row. Under these assumptions we define the weight of each partition as the number of rows multiplied with the *maximum NNZs per row* plus one for the integer indicating the number of non-zeros, and the size of $\vec{x_p}$.

$$weight = (MAX\_NNZs\_PER\_ROW + 1) \times NO\_ROWS + sizeof(x_p) \tag{4.1}$$

We now have a formula which can help us find the execution time relative to other partitions during run-time of *sparstition* . Although this is not made use of currently for reasons explained in the following section, it does offer the potential for load-balancing in the future. This model is verified in Section 7.5.1.

4.3.2 Coarse vs. Fine-Grained Load Balancing

There are two types of load balancing (LB) which can take place. The fine-grained LB takes place if each partition was not always split in half such as in Figure 4.1, but instead a parameter k is defined which stands for the partitioning point. Then we must somehow find the k where the weights are equal for the two partitions.

However, it is difficult to predict the rate of change in the sizes of the resulting $\vec{x_p}$'s. One way is to perform two *sparstition* operations, and assume linear growth of the $\vec{x_p}$'s. Then line equations are computed and the intersection is found, i.e. where the two partitions have equal weights. However, that is not compatible with every sparsity pattern as it is not guaranteed that the size of $\vec{x}$ grows linearly. Furthermore, the computation time doubles as a result of the extra *sparstition* operation, which the resulting improvement from the load balancing must justify.

Even if the execution times were adjusted, it does not guarantee that subsequent *sparstition* will have equal weights. Consider two partitions, P_1 and P_2 that have equal weights, i.e. $w(P_1) = w(P_2)$. Let us perform *sparstition* on them both which result in P_{11}, P_{12}, P_{21} and P_{22} for P_1 and P_2 respectively. Even if $w(P_{11}) = w(P_{12})$ and $w(P_{21}) = w(P_{22})$, it is not guaranteed that $w(P_{11}) = w(P_{21})$ and so forth. This becomes the case for matrices that have irregular density of non-zero columns such as with Hamrle3 in Figure 4.5.

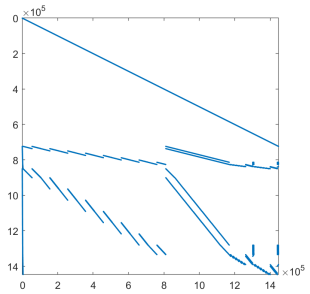


Figure 4.5: The sparsity pattern of Hamrle3.

When it is split in half, then the two $\vec{x_p}$'s are of the same size (DIM) (recall that the size is determined by the density of non-zero columns). However, when each of the partitions are partitioned resulting in $N_P = 4$, then the load balancing done previously becomes useless as the sparsity pattern in the upper half is vastly different from the lower half.

In coarse-grained LB, the weights of each partition are computed during *sparstition* and used right before the kernel is called. The partitions can be grouped according their weights and distributed evenly to accelerators. There is considerable flexibility in this LB technique as we will see in Section 7.5.2.

4.4 Conclusion

We defined the requirements for the `Sparstitioner` and have tailored our design to meet them. The result is a parameterizable framework which can partition a matrix using the *sparstition* algorithm until the memory constraint is met. We argued that the need for fine-grained load balancing was unnecessary for the target platform, and thus we always split rows into groups of equal sizes. The design is, however, capable of being extended with this feature in the future due to its bipartite nature. Figure 4.6 gives a high-level diagram of the work that we presented in this chapter.

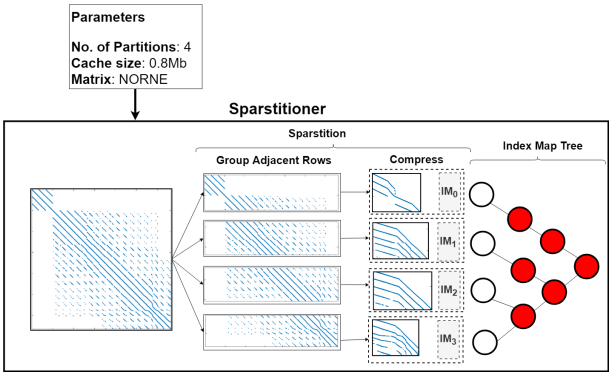


Figure 4.6: A high-level figure of the design developed in this chapter.

In the next chapter, we will look into the other component of the co-design, namely the HLS kernel. It will be designed with the goal of processing multiple partitions in an efficient pipeline.

5 HLS Design

In the previous chapter we looked into the design of the `Sparstitioner` which makes up the other component of the co-design. We will now present the design of the HLS kernel in this chapter which, together with the `Sparstitioner`, will perform partitioned SpMV. We first define the goals of the design and the kernel parameters. Then we present a high-level design of the fabric where the kernel is connected to the DRAM via DMAs. Finally, we present the HLS design and analyze the waveform to make some further optimizations.

5.1 Design Goal and Requirements

The goal of the design is to perform a number of SpMV 's as a pipelined process. This means that if a matrix is partitioned into multiple parts, each with a unique $\vec{x_p}$, the transfer of the following $\vec{x_p}$ is overlapped with the computation. This reduces the computation time by the amount of time it takes to transfer the $\vec{x_p}$'s. It follows that the first $\vec{x_p}$ must first be transferred completely before computation begins, due to the random access to it.

Secondly, the kernel must be parameterized. It would be very time-consuming to have to re-synthesize the design each time a new partitioning parameters are to be tested. However, the parameter

Finally, the design must use as much bandwidth as possible. This is because SpMV is severely memory-bound and the performance is strongly dependent on the amount of bytes delivered to the kernel each cycle.

- Pipelined design. Overlap computation with the transfer of the following vector segment.

  Transfer the first $\vec{x_p}$ before computation begins.
- Parameterized to allow for any partitioning without needing to re-synthesize the design.
- Use as much of the available bandwidth as possible

5.1.1 Kernel Parameters

The following are the parameters that must be configured before running SpMV on the kernel.

- Number of partitions
- Number of rows per partition
- Size of $\vec{x_p}$ per partition

The last two parameters are arrays of size that must be known at compile-time by the HLS compiler, and therefore a constant `MAX_PARTITIONS` is defined. This constant is accessible by the host and should be a power of two for reasons explained in Section 4.2.1.

5.2 ZYNQ Design

In order to use as much bandwidth as possible, each array of the SpMV algorithm is assigned to a dedicated port. Before the algorithm can begin, the first $\vec{x_p}$ must be cached on the FPGA. Once completed, the second stage starts which consists of both computation and the memory transfer of the following $\vec{x_p}$'s. The ports that stream $\vec{x}$, *val* and *col* values are configured to 64 bits which translates to two single-precision floating points per cycle. The *row* size values are not required as frequently. In fact, a new (integer) value is only needed every $\frac{MAX\_ROW\_SIZE}{2}$ cycles due to the zero-padding clarified in Section 5.3. The design of assigning each array to a dedicated port has been implemented before for a manually designed SpMV kernel [25].

The system is summarized in Figure 5.1.

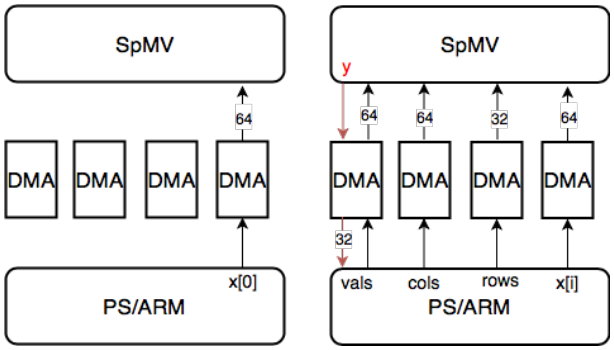


Figure 5.1: High level block design as a 2-stage process. The DMA channels are configured to either 32 or 64 bits.

Each port is connected with a DMA in *simple* mode which is faster than scatter-gather but has a limit of 2'000'000 single-precision (SP) floats. The total bandwidth is taken to be data delivered via the `vals, cols` and `rows` ports. The design is configured to run at 100MHz, `vals` and `cols` deliver 64 bits per cycle while `rows` 32-bits. The maximum theoretical bandwidth of the 64-bit ports at 100MHz is 800MB/s and 400MB/s for the 32-bit port. The total bandwidth is therefore 2GB/s.

5.3 HLS Design

The details of the HLS design will be the topic of this section. First the modulues are explained which enable the task-level pipelining of transferring subsequent $\vec{x_p}$ with the SpMV computation. Then the various directives used to optimize the design are listed and their purpose briefly explained.

5.3.1 Hardware Modules

Vivado HLS allows functions to overlap to achieve task-level pipelining. Two tasks that have been identified is the caching of the subsequent $\vec{x_p}$ and the computation of SpMV, in the case when multiple $\vec{x_p}$'s are to be cached. The functions, or modules, are therefore generated as independent components and can be executed in parallel.

5.3.1.1 The cache_x Module

The `cache_x` module is relatively simple but it simply pops two floating points from the DMA stream and caches them in one of the ping-pong buffers. If there is another `x_p` to be cached, it will then start caching to x2, then x again, and so forth. The arrays are stored in a true dual port RAM so both floats are cached each cycle.

5.3.1.2 The compute Module

The `compute` module performs the actual SpMV computation in a pipeline. It reads from the buffer written to by the `cache_x` module. The design of this module is presented in Figure 5.2.

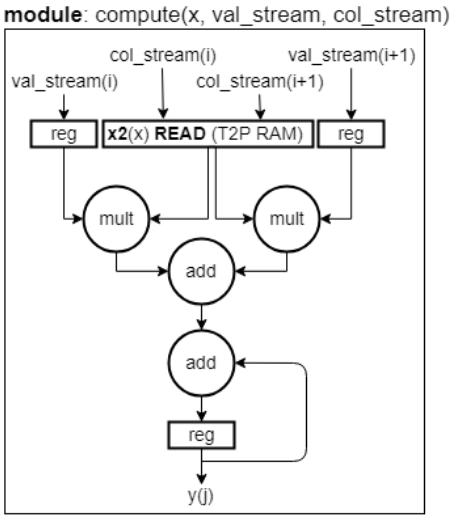


Figure 5.2: The circuit generated by the HLS synthesizer. It iterates between **reading** from `x` and `x2`.

In order to pipeline the `compute` module in , the synthesizer must be able to perform static scheduling of operations. Thus loops with an indeterminate amount of iterations are impossible to pipeline.

Recall the SpMV algorithm with CSR encoding in Algorithm 1 in Section 2.2.1, where the inner for-loop introduces the problem due to random row sizes. To solve this, the number of iterations is made constant by finding the maximum number of row elements, and zero-padding

rows smaller than the largest one. This modification results in Algorithm 2.

```
for i ← 0 to N do
    row_nnzs := row_sizes[i];
    max_row_size := 8;
    acc := 0;
    for j ← 0 to max_row_size do
        if j < row_nnzs then
            val := val[j];
            x_val := x[cols[j]];
        else
            val := 0;
        end
        acc += val × x_val;
    end
    Y[i] = accum;
end
```

As seen from Section 5.2, the kernel must be designed so that two floating points per cycle are processed. The inner loop changes to the one in Algorithm 3 as a result.

```
for j ← 0 to max_row_size do
    if j < row_nnzs then
        val := val[0][j];
        x_val := x[cols[0][j]];
        val2 := val[1][j];
        x_val2 := x[cols[1][j]];
    else
        val := 0;
        val2 := 0;
    end
    acc += val × x_val;
    acc += val2 × x_val2;
end
```

Note in the last two statements the accumulation has been kept separate. This is because if the statements were combined, the order of operations is changed which may result in a rounding error. Although this is generally not a serious issue, it makes the functional verification neater and the synthesis report reveals no significant penalty.

Now an issue arises, namely in the case of rows with an odd number of non-zeros, or more generally non-zeros that are not a multiple of the bandwidth. A dependency between iterations is created which impacts the performance significantly, i.e. clock period and latency more than doubles. This is depicted figuratively in Figure 5.3.

This is most easily solved by making every row-size a multiple of the bandwidth. This is actually handled by the Partitioner which already has the arrays in program memory, and can easily write new files which have zero-padded odd rows.

Only the `cache_x` module runs when the kernel is first started. when the first $\vec{x_p}$ is streamed to the kernel. Once that task is completed, the `compute` module starts computing by accessing the filled `x` while `cache_x` continues with caching the stream to `x2`. This continues until the last

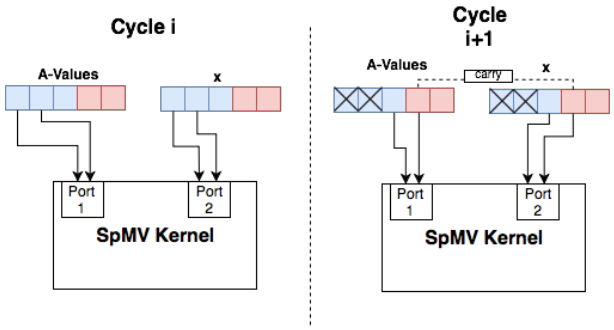


Figure 5.3: When rows have an odd number of non-zeros. Blue squares represent an arbitrary row and the red squares a subsequent one.

$\vec{x_p}$

5.3.2 Compiler Directives

In order to optimize the design, the HLS compiler offers the programmer a set of *pragma*'s which direct the synthesizer in the design space. Following is a discussion of the most common directives and an justification on why or why not they were chosen for this design.

HLS PIPELINE This is arguably the most important pragma in the design, and the reason why the problematic road of zero-padding was chosen. The outer for-loop of SpMV is assigned with this pragma.

HLS UNROLL: This one is not present explicitly, but due to the outer loop being pipelined, the inner loop is unrolled.

HLS DATAFLOW: This pragma cannot be used due to conditional execution of tasks which rises with the parameterized nature of the kernel.

HLS ARRAY PARTITION: Only $\vec{x}$ is stored in the kernel memory, and all the available BRAMs are used to store it. Since the ping pong buffers are stored in separate arrays, the only possibility for this pragma to be effective is if each value was stored in a register. This is extremely time-consuming to synthesize due to the required wiring and also completely unnecessary. There are at most two accesses to the memory each cycle, and True dual port BRAMs meet this demand.

HLS RESOURCE core=RAM_T2P_BRAM: True dual port BRAMs for reasons explained above.

5.3.3 Kernel Configuration

As the number of partitions increases, the more parameters must be passed to the kernel. Recall that for every partition, the size of each $\vec{x_p}$ and the number of rows is required. Therefore a scalable way of passing those parameters must be found to minimize the impact on performance with a growing number of partitions.

The standard way to pass parameters is to add them to the *control bus bundle* via the AXI4-Lite bus. This is a straightforward approach but the passing of parameters can not be done in parallel with the execution of the modules, so the kernel must halt until the transfer is complete.

This is not a problem when there are not many parameters to pass, but it must be addressed for the sake of scalability. Figure 5.4 is of waveform which demonstrates the blocking effect of passing the parameters.

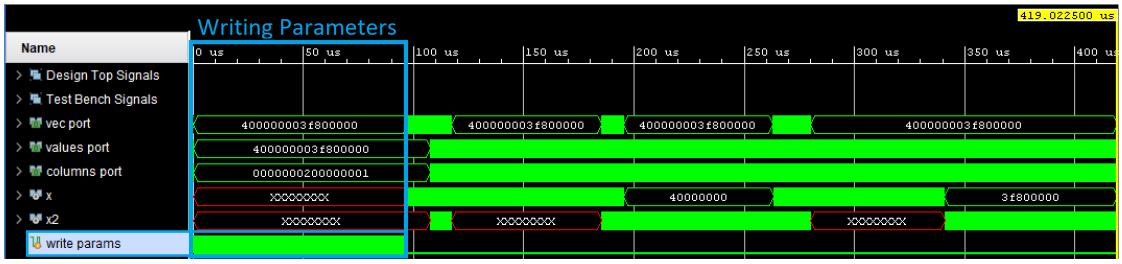


Figure 5.4: Waveform capturing the blocking effect of setting parameters on the kernel.

Recall that there are 4 ports connected to the kernels, and only one is used during the caching of the first $\vec{x_p}$. However, the size of said $\vec{x_p}$ is still needed but instead of passing every single value, only the first one needs to be passed via the AXI4-lite bus. The rest can be streamed on the other two (otherwise idle) 64-bit ports parallel to the caching module.

5.3.4 Design Analysis and Limitations

It is necessary to ensure that the kernel runs according to expectations and to also identify any unforeseen shortcomings. Figure 5.5

The Figure 5.5a has been labelled with the most significant stages which will be explained in order.

1. The transfer of parameters is complete while the transfer of $\vec{x_p}$ is still ongoing.

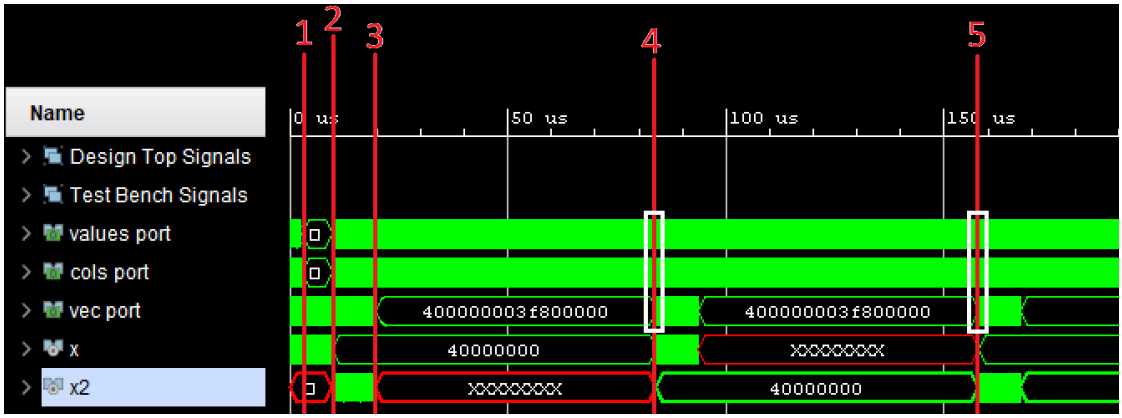
2. The transfer is complete and the second buffer is immediately written to. The first SpMV computation begins which is marked by the resuming of reading from the values and columns ports.

3. The transfer of the second vector segment completes. The first buffer is still in use so the transfer halts.

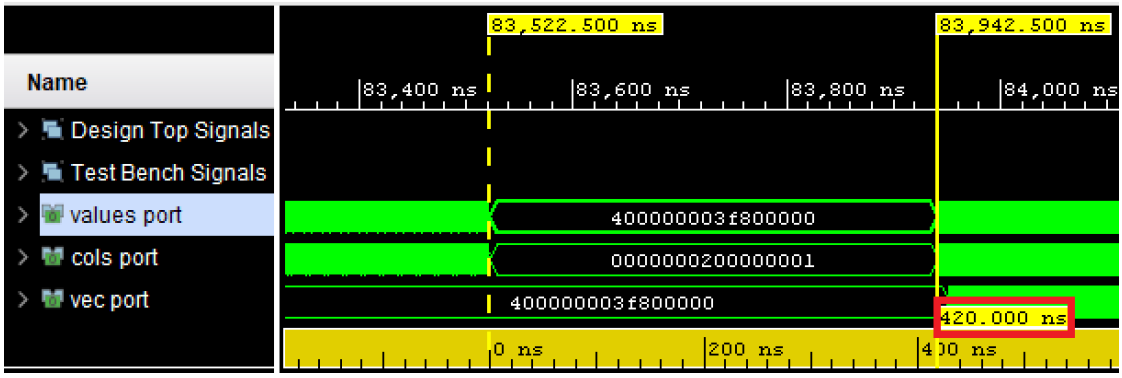
4. First SpMV computation finishes. The first buffer frees up and transfer resumes for a relatively short time.

5. Second SpMV computation finishes so the second buffer frees up. Previous step is repeated.

Notice the presence of white rectangles in steps 4 and 5 where there is a "buffer switch". These regions are magnified in the lower sub-figure which reveal some overhead in switching from one buffer to the other. This is due to the computation module having to finish its computation before the next one may begin. There is no obvious way to smoothen this transition, and it may be one of the shortcomings with the current state of HLS. However for a small number of partitions the effect is relatively small, out of the 74us between steps 4 and 5, only 420ns or 0.5%



(a) Waveform of the kernel from the beginning of execution. Significant stages have been labelled and the regions enclosed within the white rectangles are magnified in Figure 5.5b below.



(b) The magnified region within the white rectangles above. The image shows the overhead in switching between ping-pong buffers.

Figure 5.5: Part of the waveform from the beginning of execution, with the region within the white rectangles magnified in the second subfigure.

are spent on the buffer switch. However, as the number of partitions increase, more transitions will take place and the overhead will rise proportionally.

One of the limits of the HLS tool is the difficulty or even impossibility to achieve multiple pipelines. As a result, it is only possible to process one row at a time which imposes limits to the bandwidth scalability of the kernel.

5.4 Conclusion

In this chapter the design of the SpMV kernel was designed from the requirements stated at the beginning. The design is divided into two levels, one is the kernel hardware and the other is the integration to the PL fabric. The fabric design assigned a dedicated DMA controller to each port in order to maximize bandwidth. Furthermore, three out of four ports were made 64 bit wide in order to supply to single precision floating points per cycle.

The design of the kernel was made as efficient as possible by exploiting task-level paral-

lelism wherever possible and by selecting the most effective pragmas. Some shortcomings were addressed and solutions proposed such as the inability to have rows with odd number of non-zeros. The chapter concluded with a waveform analysis to verify that all the design decisions took effect.

Finally, we conclude this chapter with a schema in Figure 5.6 of the design which reveals the communication/computation pipeline which we discussed.

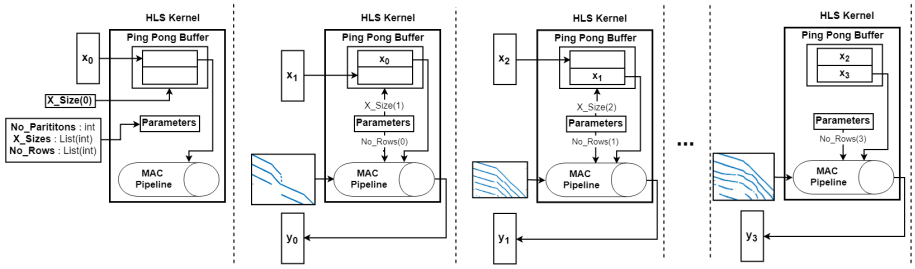


Figure 5.6: A schema of the HLS processing 4 partitions in a pipeline.

The following chapter will combine the design discussed in this chapter with the `Sparstitioner` from the previous chapter in a co-designed implementation.

Hardware/Software Co-design Implementation

6

In this chapter, the implementation details and analysis of the `Sparstitioner` algorithm are presented. `Sparstitioner` will be implemented on a regular workstation separated from the SoC in order to facilitate debugging and to keep it independent from the accelerating framework. The user may wish to port the code to the SoC, which may be very minimalistic, that is without an operating system and executing applications bare-metal. For this reason C++ is chosen, but conforming to C coding standards [1] The reason for not choosing C is so that the project can eventually move towards Object-Orientation, and to use some libraries such as the `high_resolution_clock`.

There are also subchapters devoted to the host and kernel of the Zedboard, which deal with some of the challenges were overcome during development. The host code is written in C and runs as a bare-metal application, i.e. there is no operating system involved. The kernel code is in C++ as that is the Vivado HLS standard.

6.1 Sparstitioner

6.1.1 Data Structures

The data structure for the nodes is a 1D array with a pointer to the end of the array. When a node is partitioned, the resulting two nodes are placed at the end of the array. A separate 2D array, called `binary_tree_map` keeps track of the location of each node in the binary tree. The top level, which should hold $\vec{x}$, is not part of this structure as the algorithm does not run if $\vec{x}$ will not be partitioned, and is therefore trivial. The `binary_tree_map` grows dynamically during the execution of the algorithm, and memory for a new level is only allocated when it is needed. The two arrays are depicted in Figure 6.1.

In the example shown in Figure 6.1, we are interested in accessing nodes x_3, x_4, x_5, x_7, x_8 in that order. To achieve this, the array belonging to the highest level, that is level 3, of `binary_tree_map` is read two values at a time (since each node can either have none or exactly two values). Since `binary_tree_map`[3][0] = `binary_tree_map`[3][1] = -1, the search is moved down a level by subtracting one from the level and halving the index, to access `binary_tree_map`[3-1][0/2]. This time we come across 3, which is the location of the first partition. Once there is such a hit, the search is moved again to the top level of the tree, but the pointer has moved to `binary_tree_map`[3][2], where we have the same story. The exit condition for the search is when the pointer has reached the end of the top level.

Each node is implemented as a `struct` that contains all data necessary to perform the partitioning.

```
1 typedef struct partitioning_data{
```

[1] https://www.gnu.org/prep/standards/html_node/Writing-C.html

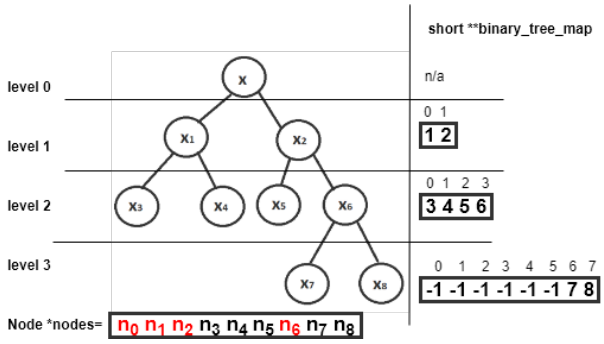


Figure 6.1: The binary tree structure mapped onto the 1D array of pointers to x_i's. Red points to memory location that has been freed.

```
2     int *index_map, *im_mapped;
3     int nnzs, no_rows, x_size, odd_rows;
4     int pre_rows,pre_nnzs;
5 } node;
```

The `Index Map` was described in detail in the design chapter and in summary keeps track of all $\vec{x} \rightarrow \vec{x_p}$ mappings. It is possible to either initialize it to -1 for non-existent mappings or to use a dedicated array `im_mapped`, discussed further in the following section.

The integer variables `nnzs, no_rows, x_size,` and `odd_rows` keep track of meta-data as suggested by their identifiers, but the variable `odd_rows` is only needed when zero-padding has been requested. The variables `pre_rows` and `pre_nnzs` are pointers to the location where each partition begins in the `col_ptrs`, `rows` and `values` arrays.

6.1.2 Analysis of the Sparstitioning Algorithm

6.1.2.1 Timing Complexity

The timing complexity of the algorithm is:

$$O(D_C \times k \times (NNZ + c_1) + (N_P + N_L) \times c_2) \tag{6.1}$$

D_C is the density of non-zero columns across all partitions. That is, there are is an extra instruction in creating a mapping in the `Index Maps` instead of retrieving a value from a mapping that exists as we see in the if-statement of Listing 6.1. Therefore, the algorithm is faster if the partition has low non-zero column density which translates to fewer mappings that need to be made. Recall that the number of non-zero columns of a partition p is analogous to the size of $\vec{x_p}$ so we can formulate

$$D_C = \frac{\sum_{p=0}^{N_P} sizeof(x_p)}{N_P}.$$

k is always 1 if the memory constraint ($CACHE\_SIZE$) is satisfied because with larger number of partitions, the *sparstition* function in Listing 6.1 is called more often, but with proportionally smaller number of rows each time. However, when the memory constraint is not met, the partition has too many non-zero columns so it must be re-*sparstitioned* as we saw in Section 4.2.1. If

Option	Extra memory (bytes)	Relative Performance	
		c1	c2
Set IM's to -1	0	+	1
Use boolean flags	N	++	$\frac{1}{4}$
Use bit flags	$\frac{N}{8}$	+++	$\frac{1}{32}$

Table 6.1: Summary of the relative cost of initializing and updating `Index Maps`

said partition contains 25% percent of all non-zeros, then k accumulates to $1+0.25=1.25$. N_P is the number of partitions, and the c_i's are the accessing (c_1) and initialization (c_2) times of the `Index Maps`. N_L stands for partitions that did not meet the memory constraint.

Listing 6.1: Code of the *sparstition* function.

```
1
2  sparstition(int *cols, const int *rows, const int *read_cols, node *nodes) {
3      for(int p = 0; p < 2; p++) {
4          node &pd = pds[p];
5          for (int r = 0; r < pd.no_rows; r++) {
6              int nnzs = rows[r + row_counter];
7              for(int nz = 0; nz < nnzs; nz++){
8                  int col = read_cols[nz + seen];
9                  bool used = pd.im_mapped[col];  //or pd.im[col]==-1
10                 if(!used){
11                     new_col = x_size++;
12                     pd.im[col] = new_col;
13                     pd.im_mapped[col] = 1;
14                 } else {
15                    new_col = im[col];
16                 }
17                 cols[nz + seen] = new_col;
18         }
19         seen += nnzs;
20     }
21 }
```

There are three options, summarized in Table 6.1, to choose from when it comes to keeping track of which mappings have been made in the `Index Map`'s, and this choice affects the c_i's. The first option is to initialize the `Index Maps` with -1's immediately after its memory has been allocated. The advantage of this approach is that no extra amount of memory is needed, and the look-up to determine if a mapping exists requires no additional logic, resulting in a small c_1. However, when `N` grows large the time it takes to set every single value becomes very time-consuming, and this needs to be done for every single partition so $N_P \times c_2$ will quickly dominate the execution time of the algorithm.

The other option is to make use of `im_mapped`, mentioned in the previous Section, of type `bool`, which is a single byte and is therefore faster to initialize than an array of `int`'s. This array is then set separately from the `Index Maps` and should in theory reduce c_2 by a fourth, but the

extra operation that sets the flag in line 13 of Listing 6.1 influences c_1 slightly.

The third and final option is to assign a bit to each mapping. The aforementioned `im_mapped` is changed to type `char` with each entry storing 8 mappings. Since `N` mappings need to be tracked, $\lceil \frac{N}{8} \rceil$ `char`'s are required which in theory results in 32x faster initialization compared to the first option and 8x speedup compared to the second. To retrieve each bit, first the corresponding byte is found by accessing the byte stored in col/8. Then a bit-wise shift operation is applied by modulo 8 which moves the target bit to the right-most position. Finally the shifted byte is masked with 1 which allows us to retrieve the desired bit. To set the bit, 1 is shifted leftwards to its position using module 8 again, and the byte stored in memory is operated with a bitwise-or. The extra bit-wise operations are performed for every non-zero and shift the weight from c_2 to c_1. These operations are shown in Listing 6.2.

Listing 6.2: Retrieving and setting a bit. **b-and** and **b-or** refer to bit-wise conjunction and disjunction. This code fits in from line 9 of Listing 6.1.

```
1 int byte = im_mapped[col / 8];
2 bool used = (byte >> (col%8)) b-and 1; //retrieving bit
3 if (!used) {
4   ....
5   //set relevant bit to 1 and store result
6   im_mapped[col/8] = 1 << (col%8) b-or byte;
7 }
```

A study of the practical performance is the subject of Section 7.3.1.

6.1.2.2 Memory Complexity

The memory required by the `Sparstitioner` will be at the very least the matrix data, i.e., the values, rows, and columns, and the `Index Maps`. There will be N_P `Index Maps` and each will require `N` values. The shuffling of column pointers can be done *in-place* and therefore spare memory. However, it is better to keep a copy of the original pointers in case the memory constraint is no met. Recall in Section 4.2.1 that when the *CACHE_SIZE* is not met, a solution is sought recursively. There it was also explained that it is efficient to always shuffle the original column pointers.

Therefore, the memory complexity of the algorithm can be at best $O(2 \times NNZ + N \times (1 + N_P))$ if the column pointers are shuffled *in-place*. However, in the current implementation this is not done, so the complexity is actually $O(3 \times NNZ + N \times (1 + N_P))$.

6.1.2.3 Sequence Diagram of the Implementation

The `Sparstitioner` is executed sequentially on a CPU, but much potential in speeding up the execution time with hardware acceleration is demonstrated in Figure 3.5. The current implementation is depicted in Figure 6.2.

In the implementation, none of the potential parallelism is exploited. However, there will still be a speedup in theory after some number of iterations due to the caching of the `Index Maps` and the reduced computation time of each smaller SpMV stream. Note that the computation of number of non-zeros is missing from this diagram, as it is not needed if the *sparstition* tasks are executed in sequence.

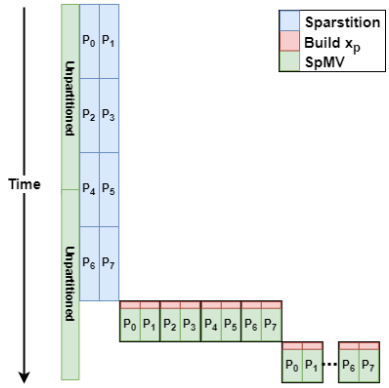


Figure 6.2: The actual sequence diagram of the tasks of the algorithm for 8 partitions. The tasks are not drawn in scale as the execution times depend on many factors such as the available bandwidth.

6.1.3 Managing Output Files

The application is run as bare-metal so simple tasks under an operating system, such as reading large files to program memory, becomes problematic. The files are kept in an SD Card which is formatted with the FAT file-system (FATFS). FATFS comes with an IO buffer of a size which must be known at compile time, and can be 256KB, 512KB and upto 4096KB. Thus there is a size limit which will eventually be surpassed when hundreds of thousands of floating points need to be stored in characters.

The `Sparstitioner` offers the user to split the result into multiple smaller files, whose size is determined by a parameter. The size of each file is tracked by accumulating the output of `fprintf`, which is the number of bytes of each line. Once the limit is reached, the file is closed and a new one is opened and written to.

Listing 6.3: Directory structure for FATFS in bare-metal applications. Directories are in bold.

```
1 -MATRIX_ID/
2     -2/
3         -cols_adj/            -v0im/               -v1im/
4             -0.ltx 1.ltx ...      -0.ltx 1.ltx ...     -0.ltx 1.ltx ...
5     -4/
6         -cols_adj/            -v0im/               ...    -v3im/
7             -0.ltx 1.ltx ...      -0.ltx 1.ltx ...  ...       -0.ltx 1.ltx ...
8     -vals_zp/      -y_gold/       -rows/      -cols/
9         -0.ltx ...         -0.ltx ...     -0.ltx ...     -0.ltx ...
```

At the top of the output directory hierarchy is the name of the matrix. This directory contains the output from a number of partitioning runs each one with N_P as an identifier. The other sub-directories are the split, and optionally zero-padded, matrix data as well as a golden $\vec{y}$ for SpMV performed on $\vec{x}$ = [0,1,2...`N` -1].

In the N_P-labeled directories, `cols_adj` and a sub-directory for each `Index Map` is present. Finally each sub-directory contains the .ltx files which have been labeled in an ascending order.

The directory structure is displayed in Listing 6.3.

6.1.4 Functional Verification

The `Sparstitioner` performs verification by computing SpMV with the normal data arrays and compares it with the *sparstitioned* results.

First, the `Index Map` is used to build $\vec{x_p}$ (see Section 6.2.2). Then the `pre_rows` and `pre_nnzs` are used as pointers to the start of the relevant partition in each matrix array. The results are then compared with a pre-computed golden result vector.

Listing 6.4: The code on the host side which builds $\vec{x_p}$ and performs partitioned SpMV. The result is then compared with a golden vector.

```
spmv(vals, rows, cols, x, res_g);
for (int p = 0; p < no_partitions; p++) {
    node &n = nodes[p];

    build_x(x, x_p, n.index_map, n.im_mapped);

    spmv(&vals[n.pre_nnzs], &rows[n.pre_rows],
        &shuffled_cols[n.pre_nnzs], x_p, y_partial[p]);

    for (int i = 0; (i < n.no_rows && !err); i++) {
        if (y_partial[p][i] != res_g[n.pre_rows + i]){
            return 1;
        }
    }
}
```

6.2 Host

6.2.1 Reading from SD Card

Before the SD card can be read, the `xilffs` library for the FAT FS must be included in the BSP (Board Support Package) settings. Also standard pin mapping must be done in the ZYNQ IP of the block diagram in Vivado.

The reading from the SD card to program memory is divided into 3 stages. First stage queries the number of files in the directory to be read from, as they may have been split up as described in the previous section. The `f_readdir` function is called until the end of directory is reached, and a counter is incremented if the pointer is on a file.

Then files are accessed following the structure in Listing 6.3 and the bytes are read into a buffer with `f_read`. The buffer is then translated into primitive types by reading bytes into a line buffer until the new-line character is read. Then the bytes of that line is cast and stored in a values array. One trick to speed up the reading of the bytes, especially for floats, is to find the shortest line. This is typically around 30 characters for double-precision, and so when a new line is being read into the line buffer the program may skip said number of bytes with a `memcpy`, thereby avoiding the conditional statement. See Listing 6.5 for pseudo-code.

Listing 6.5: Pseudo-code of bytes from the SD card cast to floats.

```
while (i < total_byte_size_of_file){
    c = (char)bytes_from_sd[i++];
    if (c != '\n') //traverse
        line_buffer[j++] = c;
    else { //cast and reset
        sscanf(line_buffer, "%d %g", &index, &values[k++]);
        j = min_bytes_per_line;
        memset(line_buffer, 0, sizeof(line_buffer));
        memcpy(line_buffer, &bytes_from_sd[i], min_line_bytes);
        i+=min_bytes_per_line;
    }
}
```

6.2.2 Building x Partitions from Index Maps

It is undesirable to do the partitioning over and over again, especially since SpMV appears in iterative algorithms such as the Conjugate Gradient, and may need to be performed hundreds of times, each time on a new vector. The matrix data always remains unchanged so it is not necessary to adjust the column pointers again, but we want to regenerate the $\vec{x_p}$'s with new values.

All the necessary information is kept in the `Index Maps` of each partition. Recall from Section 4.2 that `Index Maps` map an index from $\vec{x}$ to $\vec{x_p}$ and that there is one for every leaf of the partition tree. Before the data is written to the files, (or passed to the next stage if later the `Sparstitioner` is executed on the host), the `Index Maps` are *compressed*, such that adjacent zero-columns, or -1's, are replaced by their number. This is illustrated in Figure 6.3.

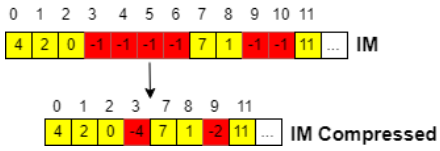


Figure 6.3: `Index Map` before and after compression.

Note that the indices are preserved and that the group of zero columns have been transformed into *empty regions*. In the implementation, an empty region is a `struct` with two pointers, the starting index and the length. Therefore the empty regions parsed out of this segment are {3,4} and {9, 2}. When the file is being parsed, each time an empty region is encountered delimited by a negative index, an empty region is created and placed in an array.

The compressed `Index Maps` are cached intermediately in files whose first line is a header which starts with a "%" symbol, and the body where every line is for a mapping. The header file consists of meta-data relative to the partition such as number of non-zeros and rows of the partition, the size of the x-vector and lastly the number of *empty regions*.

The advantage of compressing the `Index Maps` and keeping the *empty regions* in a separate `struct` is that the complexity of building $\vec{x_p}$ is only $O(sizeof(x_p))$, and there are no conditional statements present in the code shown in Listing 6.6. Note that the code is embarrassingly

parallelizable[2].

Listing 6.6: Building of $\vec{x_p}$ from `Index Maps`

```
int x_ptr = 0, start = 0, x_new_ptr = 0;
for (int i = 0; i < x_map.no_empty_regions; i++){
    empty_region_t er = x_map.empty_regions[i];

    for (int j = start; j < er.start; j++){
        int x_p_index = index_map[x_new_ptr++];
        x_p[x_p_index] = x[x_ptr++];
    }
    x_ptr += er.length;
    start = er.length + er.start;
}
```

6.2.3 Running SpMV and Functional Verification

First all the matrix data must be fetched except for the columns values, unless the programmer wants to perform SpMV on the host (recall that for the functional verification, the programmer may also use the `y_golden.ltx` file). This is because with each number of partitions in the MATRIX_ID immediate subdirectories, there is a dedicated `adjusted_cols` directory. As a result, the host code offers the possibility of running multiple *partitioning trials* in a single execution. Along with the columns, the `Index Maps` for the $\vec{x_p}$ are obtained each time a new partitioning is fetched. Before starting the communication with the kernel, one last sanity check takes place during development, namely to add up the row sizes for the partition and compare it with the `nnzs` value from the corresponding `Index Map` file. If these values are not the same, then there was an error somewhere along the line and the DMA's will hang infinitely. This check is not needed once the programmer is confident that the `Sparstitioner` is stable.

Listing 6.7: Pseudo-code from the communication with the kernel from the host-side.

```
get_matrix_data(matrix_dir, vals, rows, y_golden);

for (no_partitions = 2; no_partitions < 64; no_partitions *= 2){

    get_adjusted_columns(matrix_dir, cols, no_partitions);
    get_ims(index_maps_data, no_partitions);
    sparstition_with_im_cache(x, index_maps_data, no_partitions, N, x_partitions);

    for (p=0; p<no_partitions && err == 0; p++){

        x_vec_map_t x_map = index_maps_data[p]; //all meta-data from IM files

        Start_HW_Accelerator(x_maps); //set params

        Run_HW_Accelerator(x_partitions[p], &vals[x_map.pre_nnzs],
            &cols[x_map.pre_nnzs], &rows[x_map.pre_rows], &res_hw[x_map.pre_rows]);
```

[2] https://en.wikipedia.org/wiki/Embarrassingly_parallel

```
16
17      }
18
19      for (i = 0; i < N && !err; i++)
20        if  (y_golden[i] != res_hw[i])
21           err = 1;
22
23      free(cols);
24      free(index_maps_data);
25  }
```

Next step is to pass the parameters to the kernel via the AXI-LITE interface and send the start signal to the kernel, using libraries provided by the BSP. Recall that there are three parameters to the kernel, two of which are arrays namely the sizes of the $\vec{x_p}$'s and number of rows and the number of partitions. The functions to the arrays have an extra parameter for the length and have the word `WRITE` in the identifier and are named `XHls_accel_Write_X_SIZE_Words` and `XHls_accel_Write_NO_ROWS_Words`. The function for setting the number of partitions has the word `SET` and has the name `XHls_accel_Set_NO_PARTITIONS`. The `XHls_accel` in the identifiers refers to the *top-level function* specified in Vivado HLS. The values for all parameters are obtained from the `Index Map` files.

Finally the kernel is called with pointers to the appropriate location within the matrix arrays and the result vector. The result is checked with the golden after all partitions have finished executing.

6.3 Kernel

6.3.1 Reading and Writing with AXI Streams

As was shown in the Design chapter, each array has a dedicated port and DMA engine, which supply data to the kernel concurrently. The AXI streams are represented in the kernel with the object `hls::stream<AXI_VAL>`, that is a stream of `AXI_VAL` which are obtained by invoking the object with a *pop()* call. The `AXI_VAL` type is a `struct` from the HLS libraries and it is implemented with a C++ *template* of 4 integers. These integers specify the number of bits for various signals used by the DMA, but the most important one is the one for data, which when set to 32 is wide enough to receive one single-precision float per cycle.

The kernel code is interfaced with the DMA's in the HLS with a *top-level function*, which takes as an argument all data, including parameters, that is to be expected from the host. The signature of the function is the following:

```
1  void HLS_accel (stream<AXI_VAL> &INPUT_STREAM, stream<AXI_VAL>
       &INPUT_STREAM_COLS, stream<AXI_VAL> &INPUT_STREAM_ROWS, stream<AXI_VAL>
       &INPUT_STREAM_VEC, stream<AXI_VAL> &OUTPUT_STREAM, int NO_PARTITIONS, int
       X_SIZE[MAX_PARTITIONS], int NO_ROWS[MAX_PARTITIONS]);
```

The size of the parameter arrays must be known by the synthesizer, so the constant `MAX_PARTITIONS` is defined. The synthesizer must also know in order to generate the interface that the parameters are coming from the AXI_LITE port and that the streams are in `axis`

mode. This is directed using the pragma `HLS INTERFACE`.

The top-level function then calls the software wrapper which is a template with a few parameters. The most important one is the maximum size of $\vec{x_p}$ that the available resources allow. Note that this number must then be halved due to the *ping-pong buffer*. For the Zedboard this is only about 45'000 single precision floats. The wrapper then caches $\vec{x_0}$ and once that is done, starts the computation while streaming the $\vec{x_i}$, until it has streamed as many $\vec{x_p}$ as specified by the number of partitions parameter. The values of $\vec{y}$ are streamed out as soon as the results are available, therefore the intermediate result is only stored in registers for the multiply-accumulation operations. The result is streamed out by invoking the *write()* method of the `hls::stream` object, after having wrapped the result up in an `AXI_VAL` struct.

Doubling bandwidth

To make the most of the available bandwidth, the HP ports can be configured to supply 64 bits of data per cycle. Since some of the ports are still configured to 32 bits as per the design, two separate types are defined from `AXI_VAL`, identified obviously as `AXI_VAL32` and `AXI_VAL64`. In order to retreive the two values from the 64-bit stream, the bits are cast to a struct of two floats.

Finally, changes must be made to the block diagram in Vivado. First the appropriate HP ports are configured by opening up the ZYNQ IP block. Then the configuration of the DMA's must also be changed. If either is missing then the stream will be serialized at the neglected component into a stream of 32 bits, resulting in no bandwidth increase.

6.3.2 Functional Verification

In order to test the design it is not necessary to go through all the steps until the design can be tested in hardware. Instead, the streams can be mocked by invoking the *write* method of the `hls::stream`, and followed by a call to the top-level function. As a result, the waveform can be generated and analyzed.

6.4 Conclusion

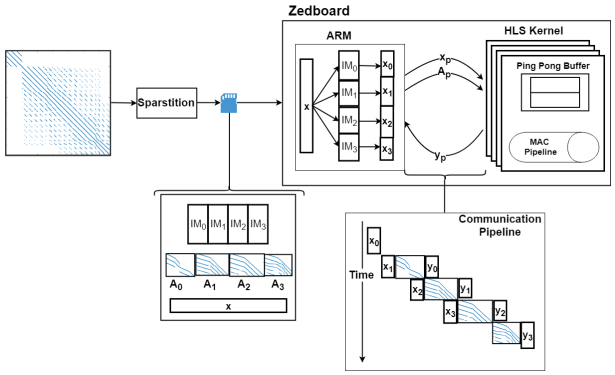


Figure 6.4: The co-design from a high-level.

In this chapter the implementation details of `Sparstitioner` were explored and an analysis of the algorithm's complexity developed. Two of the bottlenecks were the *sparstitioning*, but

also the initialization of the `Index Map` arrays. To address that, three solutions were presented and proposed which will undergo practical evaluation in the following section.

The code for ZYNQ, both host and kernel, was also explained and solutions to, for example, large file sizes and doubling the bandwidth were presented.

Now we have presented the implementation of the two components that make up this work. Their interplay is shown in Figure 6.4.

The matrix is first run through the `Sparstitioner` which produces `Index Maps` and matrix data partitions. These are written to .ltx files which are placed on an SD card. The SD card is read by the ARM processor which then executes the partitions located therein with minimal user interference.

This co-design will be evaluated extensively in the next chapter. First the focus will be on the HLS kernel as a standalone component. Then we move into the *sparstition* algorithm and evaluate the three implementation identified in this chapter. Finally, we evaluate the whole system as a whole and compare it with the state-of-the-art.

7 Experimental Results

This chapter presents the results from the testing of the co-designed implementation and how these results were obtained. The goal of this chapter is depicted in Figure 7.1.

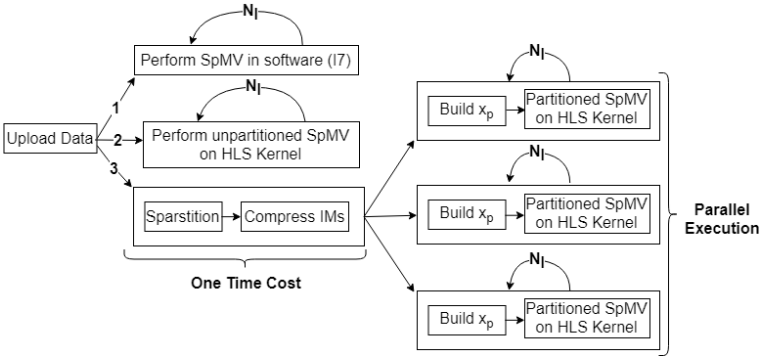


Figure 7.1: The goal of this chapter, namely to evaluate when it is optimal to compute SpMV by taking path 3 compared to paths 1 and 2.

The matrix data is first loaded into program memory and then there are three ways to compute SpMV. Once the experimental setup has been explained and the benchmarks introduced, we explore the performance of running the algorithm in software (1) and on the HLS kernel (2) without performing the *sparstition* step. The performance of the HLS kernel is also compared with the current state-of-the-art.

Then we evaluate the cost of *sparstition* in detail which only needs to be performed once but enables the computation of SpMV in parallel. The partitions are then executed in a pipeline and *in isolation* which mocks parallel computation in environments where multiple accelerators are available. The number of iterations are computed in order to find the point where the *sparstition* cost is covered, and whether it is a reasonable number compared with numbers obtained from running Bi-CGSTAB , an advanced solver. We then briefly explore the flexibility of the solution with a discussion of load balancing when the number of partitions (N_P) is greater than the number of accelerators (N_A).

Finally, this chapter closes with a discussion of the limiting effect of the bandwidth and the current state of HLS.

7.1 Experimental Setup

7.1.1 Platform

The *sparstition* algorithm runs on a workstation and the resulting files are uploaded to an SD card. The ARM reads these files and uses them to communicate with the FPGA kernel. As such,

two different platforms deliver the final result of the co-design as shown in Figure 7.2.

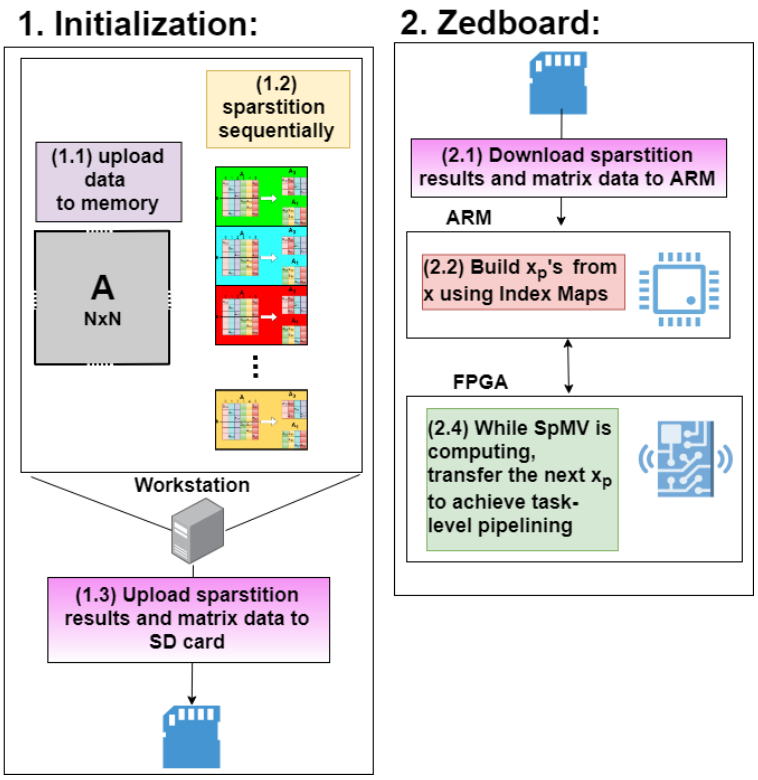


Figure 7.2: The experimental setup with the workstation and the Zedboard.

The first one is a Windows 10 workstation running with an Intel I7-8550U processor with a base frequency of 1.8GHz and 16GBs of RAM. The workstation times the algorithm with the `high_resolution_clock` from the `chrono` library, where the timer measurements are obtained right before and after the execution of the target function. The timing for *sparstition* is reported as an average of 1000 runs for the smaller benchmarks and 100 runs for the larger benchmarks. The reason is that when large benchmarks were averaged for 1000 runs, the results varied dramatically when reproduced due to noise from the operating system. Running a fewer number of tests gave much more consistent results between runs. The *sparstition* algorithm exhibits much potential for parallel execution and immediate speedup should be achievable with OpenMP. However the partitioned SpMV algorithm is run sequentially in this thesis due to us only having one FPGA at our disposal and the goal of this thesis to develop a proof-of-concept.

The Zedboard encases the ZYNQ-7020 chip that is made up of an ARM Cortex-A9 microprocessor and Xilinx Series 7 FPGA fabric. The ARM processor runs traditional C binaries and controls the FPGA fabric via the interconnect discussed in Section 2.3.1. The application runs bare-metal, in other words no Operating System is deployed, which makes the user responsible for memory management. Also some functionality such as reading from files, is made more difficult as discussed in Section 6.1.3. However, bare-metal applications are faster to deploy and no overhead is caused by an operating system. The ARM processor has a frequency of 667MHz and the FPGA fabric is configured to run at 100MHz for our experiments.

Listing 7.1: Timing on the ARM processor within the ZYNQ chip.

```
XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
init_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
curr_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
calibration = curr_time - init_time;

XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
begin_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
for (i = 0; i < NUM_TESTS; i++) {
  //run either hw or sw function here
}
end_time = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);
run_time_hw = (end_time - begin_time - calibration)/NUM_TESTS;
```

There is a dedicated IP module to perform timing measurements in AXI, namely the `AXI Timer`. This module keeps a value internally which is incremented each cycle. The difference in this value before and after the execution of the function therefore gives the execution time as a number of cycles. Since the fabric clock runs at 100MHz, the cycle count converts to seconds when multiplied by 1×10^{-8}. The timer should always be reset first to avoid overflow of the counter, and calibrated by measuring the cycle counts involved in obtaining the values. Functions that run either in hardware or software make use of this method of timing measurements. Therefore, the SpMV running on ARM and the building of $\vec{x_p}$'s is also timed in this fashion. Listing 7.1 shows the code involved in timing within the ZYNQ chip. When the SpMV results on hardware are presented in this chapter, they **always include the transfer time involved in streaming data to and from the kernel**.

Timing measurements from both the ARM and the workstation of building the $\vec{x_p}$s are included, even though technically it should only be performed on the host communicating with the kernel. The ARM microprocessor is extremely weak in performing multiple memory operations as will also be highlighted in its performance of the SpMV and therefore gives disappointing results and even becomes the bottleneck.

FLOPS (Floating Point Operations per Second) are taken to equal $2 \times NNZ$ in this work. Lastly, bandwidth is measured by removing the caching of $\vec{x}$ and the MAC (Multiply-Accumulate) module in Section 5.3.1 is transformed to a *copy-loop*, i.e. the input is connected directly to the output. The number of bytes in the matrix data arrays is then divided by the execution time. The theoretical maximum bandwidth in our design is 2GB/s as discussed in Section 5.2.

7.1.2 Maximum Vector Size

There are 280x18Kbit BRAMs available on the ZYNQ chip. This translates to 630KBytes which can store 160'000 single precision floating points. Since we divide this number in two for the ping-pong buffer which we discuss in Section 5.3.1, the theoretical maximum $\vec{x_p}$ size is 80'000. However, the maximum size is 32'000 in practice once the kernel is parameterized. Since the kernel design synthesizes in Vivado HLS but not during the synthesis of the block level design. The cause is unknown but memory may be needed for the FIFOs of the DMAs.

7.1.3 Benchmarks

All of the chosen benchmarks except for NORNE and Hummocky are taken from the University of Florida Sparse Matrix Collection[26]. The benchmarks chosen to test the co-design come from a wide variety of domains. Furthermore, they are divided into two categories depending on whether the $\vec{x}$ fits in cache or not (recall that the matrix data is streamed and only the vector must be stored).

Table 7.1: Benchmarks used to verify the *sparstition* algorithm and the HLS kernel. Starred benchmarks must be partitioned to be performed on the Zedboard.

Matrix	N	NNZ	Largest Row	Size of N Relative to Cache	Minimum N_P	Application Domain
Hummocky	12,380	120,058	11	0.077	2	Oil Reservoir
epb1	14,734	95,053	7	0.092	2	Thermal Dynamics
wathen100	30,401	471,601	21	0.19	2	Random 2d/3d problem
dixmaanl	60,000	299,998	6	0.38	8	Optimization Problem
epb3	84,617	463,625	6	0.53	4	Thermal Dynamics
NORNE*	133,293	2,776,851	57	0.83	16	Oil Reservoir
Lin*	256,000	1,766,400	7	1.6	16	Eigenvalue problem
parabolic_fem*	525,825	3,674,625	7	3.29	128	Fluid Dynamics
roadNet-PA*	1,090,920	3,083,796	9	6.8	64	Road Network
Hamrle3*	1,447,360	5,514,242	6	34.46	512	Circuit Simulation

The largest row is an important metric for HLS to determine the depth of the pipeline during scheduling of operations. The rows that are smaller are zero-padded , which is a requirement of HLS to create pipelined circuits. Note that this is at the same time also a drawback of HLS. The N_P is determined by size of the largest $\vec{x_p}$ as they must all fit within the available cache of the ZYNQ. Due to the effect of parameterizing the design on the memory available discussed in the previous section, some matrices such as epb3 which fit in the cache, cannot be performed for $N_P = 2$. The column with the relative size values is computed with the largest theoretical $\vec{x}$ size, namely 160'000 single precision floating points.

7.2 HLS Performance

7.2.1 Resource Utilization

Table 7.2 sums up the total resource usage of the design.

Table 7.2: The resource utilization of the design

	Number Used	Percentage Used
BRAM	132	47
DSP48	10	4
Flip Flops	13215	12
LUTs	9605	18

We attempt to allocate as much memory as possible for the ping-pong buffers which explains their high usage. Ideally, would therefore prefer 100% and Vivado HLS allows us to allocate more BRAMs. However, in Vivado when the entire design is synthesized we run into problems.

We assume that buffers need to be allocated for DMAs which use FIFOs to interface with the kernel. The Vivado HLS tool reports that 2 DSP48s are used to implement an adder, and 3 are used to implement a multiplier. This corresponds directly with our design in Figure 5.2 as it has two adders and two multipliers, which results in 10 DSP48 blocks.

7.2.2 Results for Benchmarks

The kernel achieves performance between 266 and 309 MFLOPS for the small benchmarks that can fit in the cache of the FPGA. This translates to a speedup of 109x - 150x compared to the ARM processor and 0.52x - 0.67x compared to Intel I7. These results are summarized in Table 7.3.

Table 7.3: Performance of the HLS kernel for the smallest benchmarks compared with the performance of Intel-I7 and ARM.

Matrix	**Performance (MFLOPS)**	**Bandwidth (GB/s)**	**Speedup**	
			ARM	Intel I7
Hummocky	287.09	1.31	119.64	0.52
EPB1	273.61	1.32	117.56	0.53
wathen100	266.06	1.15	109.83	0.46
dixmaanl	281.2	1.44	135.82	0.57
epb3	309.7	1.57	150.51	0.67

7.2.3 Bandwidth Scalability

The performance scales with bandwidth until the maximum row size is delivered each cycle. The maximum row size of each benchmark is in the *Largest Row* column of Table 7.1. In the current design, at most 2 values and column pointers are delivered to the kernel each cycle. The kernel scales with bandwidth up until the point where it can deliver a row of the maximum size each cycle. Beyond that, the issue of maintaining multiple parallel pipelines becomes an issue which we assume is not currently possible. We also assume that **only a single row** is delivered each cycle at most, and not a part of the following row as well. This is due to the issue discussed in Section 5.3.1.2.

Consider dixmaanl that contains at most 6 non-zeros per row. Currently, a row is received every $\frac{6}{2} = 3$ cycles, and it would achieve maximum speedup with the current design if the bandwidth increased 3-fold. Therefore, the maximum achievable performance with the current design, when scaled for bandwidth, is $281.2 \times 3 = 843.6 MFLOPS$. Table 7.4 completes the results.

Of course, the P_{PEAK} becomes more unrealistic as the *Largest Row* increases. An additional issue that we have not dealt with, due to BRAMs having 2 True Dual ports, is the limited number of ports for memory access. The data would need to be organized in the BRAMs in such a way that not more than two from each are needed each cycle. Recall that the ZYNQ chip contains multiple BRAMs and it is possible to organize how the data is spread across them. This may be a significant challenge.

Table 7.4: The theoratical peak performance P_{PEAK} of the HLS kernel when scaled for bandwidth.

Benchmark	P_{PEAK} (MFLOPS)	Largest Row
Hummocky	1579.00	11
epb1	957.64	7
wathen100	2793.00	21
dixmaanl	843.60	6
epb3	929.10	6

7.2.4 Comparison with State-of-the-Art HLS

The kernel is compared with the HLS design in [3] by running the same benchmarks with double precision floating point. These are the only results that are in double precision but the kernel functionality is still essentially the same, except a single double precision floating point is streamed each cycle instead of two single precision floating points in the values and vector ports. Furthermore, the results from the work are **from simulation** only whereas ours are actual hardware results including transfers. The result from this comparison is summarized in Table 7.5.

Table 7.5: Comparsion of this work with [3] for their benchmarks in double precision. Largest row is analogous to pipeline depth. The bandwidth refers to this work as [3] reports performance in simulation.

Matrix		Execution Time (ms)			
Name	Largest row	[3]	This work	Bandwidth (GB/s)	Speedup
bcsstm25	6	2.8	2.15	0.29	1.3
dw8192	8	3.5	0.77	1.02	4.6
bcsstk12	27	1.0	0.46	0.68	2.2
ex7	75	3.0	1.29	0.70	2.3
poli3	336	5.5	N/A	N/A	N/A

Unfortunately the workstation does not have enough RAM to synthesize the design necessary for the `poli3` benchmark. This is due to the vast amount of scheduling required in order to have a pipeline of the required depth.

7.3 Sparstition Performance

The *sparstition* algorithm can be broken down into 2 major steps, initializing the `Index Maps` and the *sparstition* function itself. Section 6.1.2 discusses three implementations of the former step and the trade-offs associated with them. This section demonstrates these trade-offs for a small, a medium, and a large benchmark. These three implementations are referred to as:

- `Index Map` implementation.

- Boolean map implementation.
- Bitmap implementation.

One consideration that is not to our advantage is that the *sparstition* algorithm is run sequentially as noted in Figure 6.2. However, with multiple nodes it becomes possible to parallelize *sparstition* as noted in Figure 3.8. The results presented in this chapter thus have a great potential for improvement.

The cost of compressing `Index Maps` is finally explored and it is factored in to the total cost of the algorithm. Even though it costs extra time to perform this action, the speed-up gained in building the $\vec{x_p}$'s quickly pays off.

7.3.1 Cost of Each Implementation

Recall that we derived the time complexity of the algorithm in Section 6.1.2 to be as follows:

$$O(D_C \times k \times (NNZ + c_1) + (N_P + N_L) \times c_2).$$

Where D_C is the density of non-zero columns across all partitions, k is a factor for how many non-zeros have been mapped, N_P is the number of partitions, the c_i's are the accessing (c_1) and initialization (c_2) times of the `Index Maps` and finally N_L stands for partitions that did not meet the memory constraint. For our experiments, $k = 1$ since the memory constraint is never violated so each non-zero is mapped only once. Consequently, N_L is always 0.

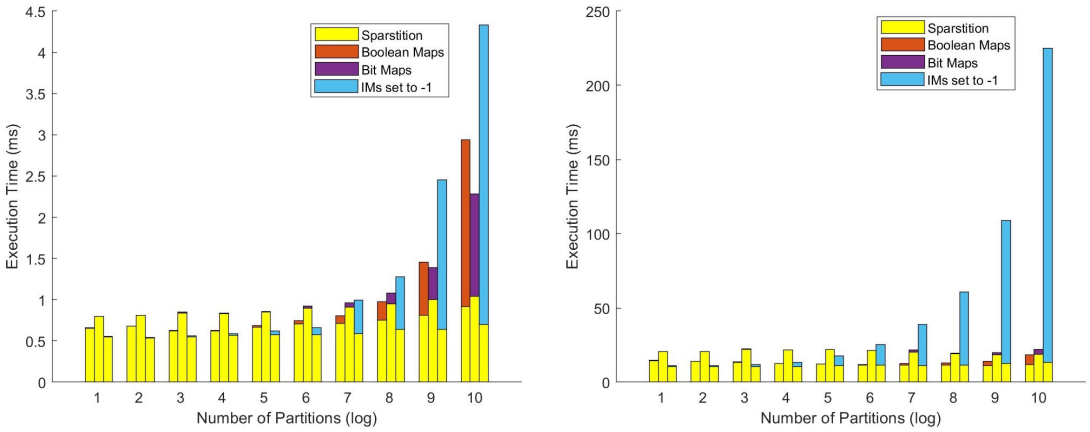
To briefly summarize, the fewest amount of instructions are executed when the `Index Maps` are initialized to -1, but this does not scale because `N` integers must be set for every partition. Boolean maps have only a single extra instruction when it sets the value but have `N` Booleans to set for every partition. Finally, bitmaps initialize the lowest number of values, or `N` bits for every partition, but in turn requires the largest number of instructions.

Notice in Figure 7.3 that for all matrices, the *sparstition* remains constant independent to the number of partitions. The `Index Map` implementation is competitive for small number of partitions but grows at the fastest rate as per the prediction. bitmaps grows at the slowest pace and is the least affected by the size of the matrix. However, the *sparstition* takes more time so for smaller number of partitions, the boolean maps are usually faster. Once compressing the `Index Maps` is factored in, the bitmaps start dominating again as will be demonstrated in the following section.

7.3.2 Compressing Index Maps

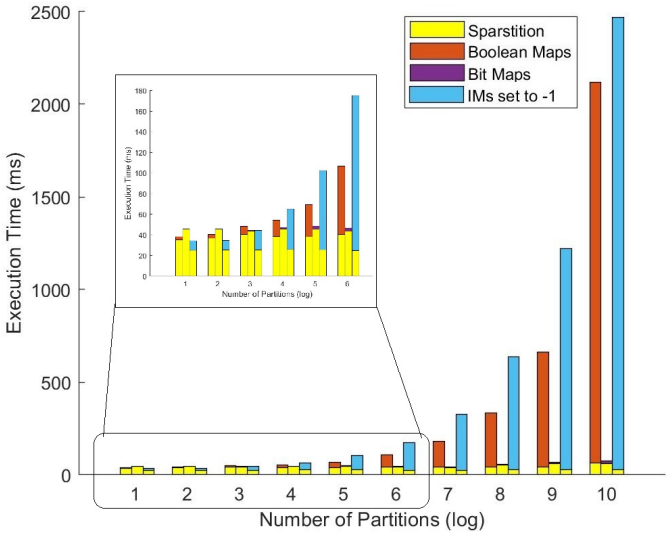
As the number of partitions increases, the `Index Maps` become more and more sparse. Without compression, the algorithm must iterate through `N` elements which will increasingly consist of empty mappings as the number of partitions increases. The compression basically takes adjacent empty mappings and replaces them with a single number indicating how many they were which are referred to as `empty_regions` in Section 6.2.2.

There is added cost to perform the compression of `Index Maps` so there must be speedup gained when building $\vec{x_p}$. Recall that building of $\vec{x_p}$ occurs multiple times in iterative algorithms. The cost for the three benchmarks are summarized in Table 7.6.



(a) Hummocky (small)

(b) NORNE (medium)



(c) Hamrle3 (large) with column clusters 1-6 magnified.

Figure 7.3: Small, medium and large test cases for all implementations. The bars depict the boolean maps, bitmaps and setting `Index Maps` to -1 in the order from left to right. The x-axis is on a logarithmic scale.

The bitmap implementation offers a faster way to perform the compression. Recall that each bitmap is an array of `chars`, so instead of checking if every single bit is set to 0, a faster way is to check if the whole `char` is set to 0. If so, the algorithm jumps over 8 bits but otherwise checks each one. The code is shown in Listing 7.2. The building of $\vec{x_p}$s with the `CIMs` is the same for all implementations.

Table 7.6: Table summarizing the cost in time of performing the compression of `Index Maps` for bitmaps and the other (boolean maps and setting to -1) implementations. The result from adding the bold numbers with the `CIM` (compressed `Index Map`) build time is lower than the build time with the sparse `Index Maps` . All times are in milliseconds.

Number of Partitions (log)	Hummocky				Norne				Hamrle3			
	Build x_p w/ sparse IM	Compressing IMs			Build x_p w/ sparse IM	Compressing IMs			Build x_p w/ sparse IM	Compressing IMs		
		Compression		Build x_p with CIMs		Compression		Build x_p with CIMs		Compression		Build x_p with CIMs
		Bitmaps	Other			Bitmaps	Other			Bitmaps	Other	
1	0.10	0.11	0.09	0.05	1.11	1.14	1.23	0.54	14.28	17.87	14.22	16.61
2	0.15	0.13	0.16	0.05	1.66	1.53	2.16	0.44	21.42	22.80	26.65	17.25
3	0.29	**0.16**	0.27	0.06	3.19	**2.26**	3.76	0.58	34.36	25.39	56.89	17.07
4	0.54	**0.22**	0.49	0.07	6.29	**3.77**	6.67	0.88	65.11	**31.54**	104.30	17.98
5	1.08	**0.46**	1.14	0.08	12.18	**5.55**	12.16	1.06	122.47	**40.61**	204.24	16.85

Listing 7.2: A property of bitmaps allows for faster compression.

```
char byte = pd.im_mapped[i / 8];
while (byte == 0){
    i += 8;
    byte = pd.im_mapped[i / 8];
}
```

The speedup gained from building $\vec{x_p}$'s with `CIM`s justifies the compression time for all cases especially in iterative algorithms where the step must be executed multiple times. Bitmaps also prove to be much more scalable in performing the compression for all benchmarks. Only `CIM`s will be tested for the remainder of this chapter since the compression results in much faster build times.

7.3.3 Total Execution Time

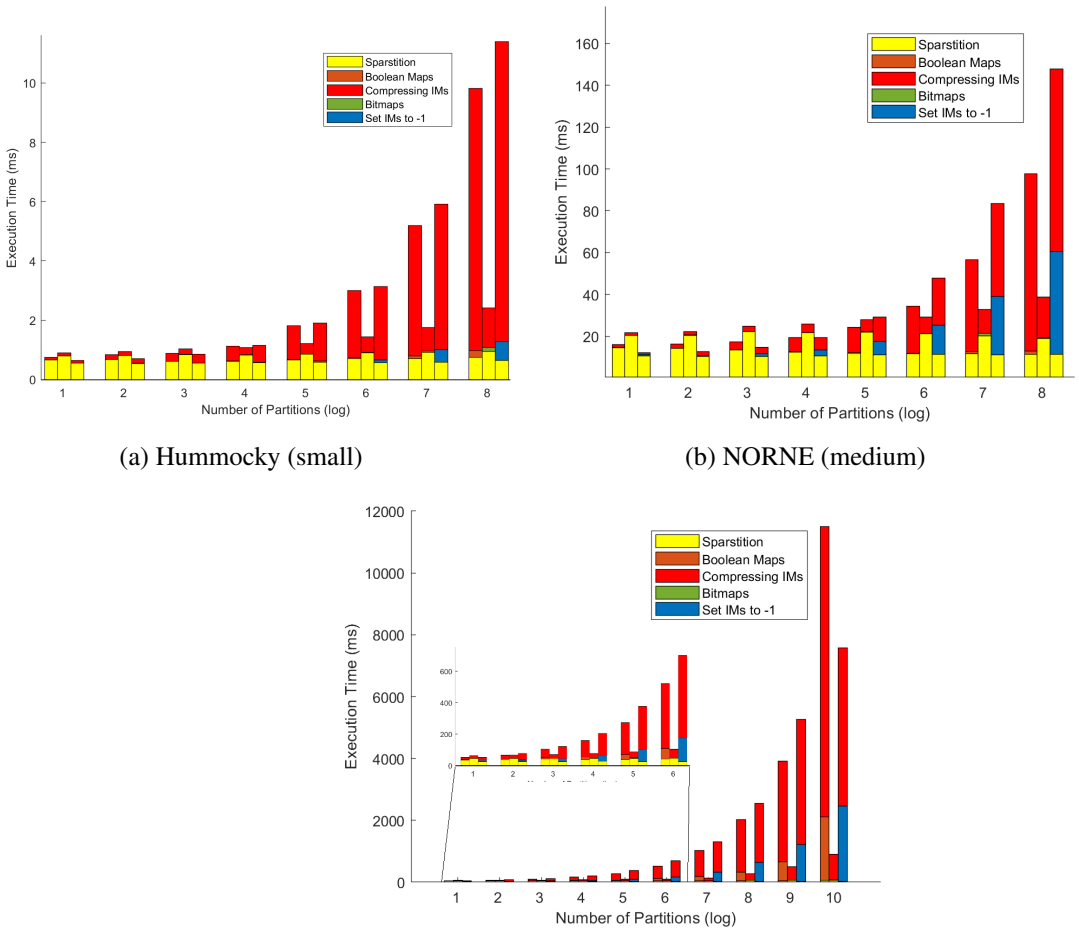
To summarize, the total execution time consists of these three steps:

1. Initializing the `Index Maps` or supplementary arrays (boolean or bits).
2. The *sparstition* algorithm.
3. Compressing the resulting `Index Maps` (`CIM`).

Building of the $\vec{x_p}$s is not included here because it takes place during the computation on the host machine. The three steps and their relation to one another is illustrated in Figure 7.4 for the three benchmarks.

It is clear from the bar chart in Figure 7.4 that the compression dominates the total execution time. But as we have seen in the previous section, it quickly results in faster build times of $\vec{x_p}$'s especially when iterative methods are considered. When a larger number of partitions is required, bitmaps is clearly the best choice as seen by the middle bar in each cluster. In the case of a small number of partitions, the best choice depends on the matrix at hand. Figure 7.5 contains three graphs, each showing the total execution time with a varying number of partitions.

For all graphs, the intersection with bitmaps (blue curve) marks the point where the slower *sparstition* time has been compensated by the faster initialization time. Where this intersection takes place varies with each benchmark but factors such as the density of `Index Maps` could be investigated in order to build a model. Unfortunately, this is not explored in this thesis.



(a) Hummocky (small)

(b) NORNE (medium)

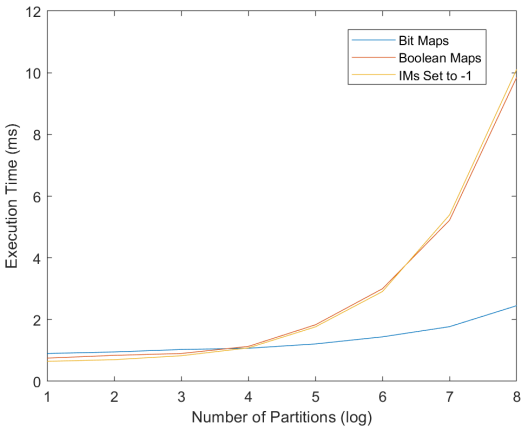
(c) Hamrle3 (large)

Figure 7.4: Small, medium and large test cases for all implementations including `CIM` time.

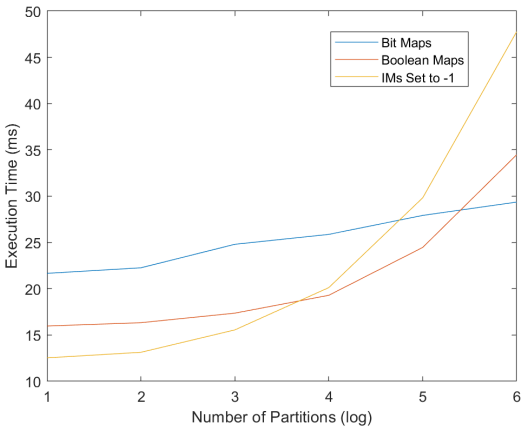
The graph in Figure 7.5a shows a narrow gap in performance between the `Index Map` and boolean map implementations. The `Index Map` implementation is faster with the smallest number of partitions by a small margin, and the margin remains narrow after the intersection between 2^3 and 2^4. For NORNE in Figure 7.5b, the `Index Map` implementation is considerably faster for the first few number of partitions but is eventually overtaken by boolean maps and finally the bitmaps. In the last Figure which depicts Hamrle3, the `Index Map` implementation never results in the fastest execution time. The gap between the `Index Map` and boolean map implementation is larger than in Figure 7.5a and stays relatively constant.

7.4 Sparstitioned SpMV

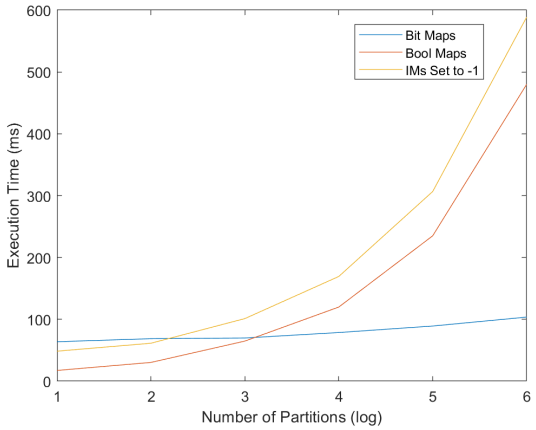
In this section, the results from the *sparstition* algorithm are combined with the execution times for partitioned SpMV .



(a) Hummocky (small)



(b) NORNE (medium)



(c) Hamrle3 (large)

Figure 7.5

First, we will explore the effect on performance as an increasing number of partitions are pipelined on a single accelerator. This is first performed on the smallest benchmarks, which fit entirely in the available cache. Subsequently, we construct a model in order to estimate the performance of the matrices that need to be partitioned more than once in order to fit.

Second, we emulate the situation where we have multiple accelerators by computing each partition *in isolation*. We claim that by taking the largest execution time will provide an adequate estimate of running the partitions on multiple machines.

Finally, we emulate the situation where multiple partitions are assigned to an array of accelerators. This is ensued by a demonstration of the platform's potential for load balancing.

7.4.1 Pipelined Execution

The *sparstition* algorithm can be used when a single accelerator is available but the matrix is too large to fit in the cache. The pipelined design masks transfer of the multiple $\vec{x_p}$'s by perform-

ing the computation of SpMV in parallel as discussed in the **HLS Design** chapter. Figure 7.6 illustrates the effect on performance as the number of partitions increases for the three smallest benchmarks.

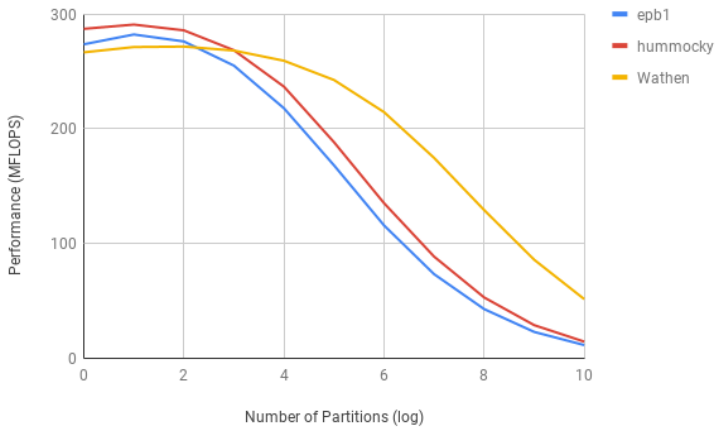


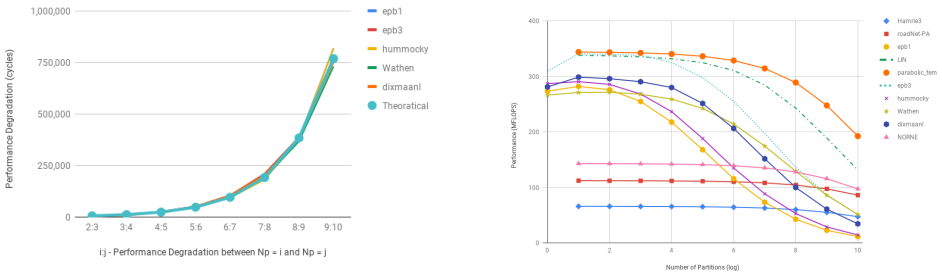
Figure 7.6: The change in performance of the smallest 3 benchmarks as N_P increases.

Notice that there is initially an increase in performance relative to the non-partitioned case. This is due to the pipelining which masks the transfer of the second half of the vector and therefore the effect is most noticeable for epb1 which is a pure bandwidth matrix, i.e. non-zeros are only near the diagonal. The performance then levels off and then rapidly degrades to almost negligible performance. This is due to the overhead in switching between the ping-pong buffers identified in Figure 5.5. As the number of partitions increases, the number of switching between buffers increases proportionally and the overhead quickly multiplies. Since the number of partitions doubles every iteration in our testbench, the overhead doubles as well.

The *Minimum* N_P column of Table 7.1 indicates the minimum N_P in order to compute each benchmark on the ZYNQ. The predictable overhead discussed in the previous paragraph and in Section 5.3.4, permits us to derive the estimated performance for the missing partitions for the larger benchmarks by halving the difference between two pipelined executions that exist. So for example, consider a benchmark takes 1'000'000 cycles with $N_P = 32$ and 1'500'000 cycles with $N_P = 64$. The difference of 500'000 cycles halves, in theory, when moving from $N_P = 32$ to $N_P = 16$. Therefore, we estimate that it takes 1'000'000-250'000 = 750'000 cycles to compute a pipelined execution of 16 partitions for this particular benchmark. This theory is tested against actual execution times of the smaller matrices in Figure 7.7a. The actual effect on performance is in the accompying Figure 7.7b.

Table 7.7 sums up the rest of the benchmarks missing from Table 7.3 by using the model to derive theoretical performance values for 2 partitions.

The maximum amount of data that can be transfered via DMA configured in simple mode is 8MBs. This translates to 2'000'000 SP floating points which is an issue for most benchmarks presented in Table 7.7. Recall that $\vec{x}$ is not stored in bandwidth measurements but the values and column pointers still had to be streamed. The solution was to split the arrays into batches and stream them *in isolation*. The time was accumulated and used to represent as if the whole data arrays had been streamed in one transfer.



(a) The difference in measured execution times between batches of different number of partitions. The theoretical curve assumes that the difference doubles.

(b) The effect on performance for all benchmarks including those derived from the model in Figure 7.7a.

Figure 7.7: The pipelined execution times for all benchmarks including those that could not be computed on hardware due to size constraints.

Table 7.7: Results from the larger half of the benchmarks. These performance values are derived and are theoratically achievable with a larger cache size.

Matrix	**Performance (MFLOPS)**	**Bandwidth (GB/s)**	**Speedup**	
			ARM	Intel I7
NORNE	143.04	0.59	54.52	0.24
Lin	338.14	1.47	142.90	0.62
parabolic_fem	344.08	1.49	148.54	0.63
roadNet-PA	112.33	0.53	58.35	0.39
Hamrle3	66.09	1.14	98.06	0.50

7.4.2 Isolated Execution

Isolated execution means that the pipelined functionality of the kernel is not activated, so instead the time to compute is measured for each partition individually. This method emulates environments were multiple accelerators are present, which requires us to define assumptions as we do not possess all the required information to make an accurate model.

The cost of performing the *sparstition* and `CIM` was covered in detail in the previous section, and thankfully it is a one-time cost. The advantage is that the SpMV has been split into independent streams which can be computed in parallel in shorter time but it may take a number of iterations (N_I) before speed-up is achieved. This is illustrated in Figure 7.1.

In this sub-chapter, we will explore how large N_I must be until the *sparstition* cost is justified. We then compare this number to an actual iteration count obtained from running Bi-CGSTAB to compute the theoretical speedup.

7.4.2.1 Assumptions

The first assumption is that the partition that takes **the longest to compute is the total execution time of the cluster**. This is because all partitions are assumed to run in parallel and the total time of the cluster therefore depends on the accelerator that finishes last. The computation

time includes both the building of $\vec{x_p}$ and the SpMV execution itself. The largest N_P in our experiments is 1024 and it is unrealistic to expect such a number of accelerators to execute all partitions in parallel. However, the point is to show the scalability of the solution, and as we will see in the subsequent section, it is possible to group partitions together and assign each group to an accelerator.

The second assumption is that $\vec{x}$ **arrives at each accelerator at exactly the same time**. This requires the driver node to have high bandwidth in order to stream the vector simultaneously to a number of nodes, and again becomes more unrealistic with an increasing number of partitions.

The third assumption is that there is **no overhead in the iterative algorithms to commence parallel SpMV** . This overhead is minimized with a deep task-level pipeline of the iterative algorithm and therefore is most viable if it is implemented on FPGAs. It depends on the implementation and the platform how efficient this pipeline is.

7.4.2.2 Computing Iterations until Speedup

Once all times have been added up, it is usually the case that a speed-up is not observed. However, as has been noted on multiple occasion, the SpMV has no practical value when performed by itself but is a vital step in many iterative algorithms. Recall from Figure 7.1 that it is in our interest to calculate how many iterations are necessary until speedup is observed. Table 7.8 summarizes how the number of iterations are computed for epb1 as an example case.

Table 7.8: Table summarizing total execution times for epb1. Only the bitmaps implementation is considered in this table for simplicity. NP HLS stands for non-partitioned HLS. All times are in milliseconds.

Ref. Execution Times		**No. of Partitions (log)**	**Total Sparstition Time,** T_S	**Build** $\vec{x_p}$		**Partitoned SpMV,** T_{PS}	**Iterations until Speedup Against**			
							T_{R1}		T_{R2}	
I7, T_{R1}	**NP HLS,** T_{R2}			**I7,** T_{Bi}	**ARM,** T_{Ba}		**Build** $\vec{x_p}$ **on ARM,** T_{IHa}	**Build** $\vec{x_p}$ **on I7,** T_{IHi}	**Build** $\vec{x_p}$ **on ARM,** T_{IIa}	**Build** $\vec{x_p}$ **on I7,** T_{IIi}
0.37	0.69	1	0.66	0.02	5.91	0.36	x	x	x	3
		2	0.68	0.01	3.17	0.20	x	5	x	2
		3	0.71	0.01	1.72	0.11	x	3	x	2
		4	0.78	0.01	0.99	0.07	x	3	x	2
		5	0.90	0.004	0.65	0.05	x	3	x	2
		6	1.11	0.004	0.46	0.04	x	4	6	2
		7	1.48	0.002	0.32	0.04	116	5	5	3
		8	2.10	0.002	0.17	0.03	13	7	5	4
		9	3.44	0.002	0.09	0.03	14	11	7	6
		10	6.48	0.001	0.05	0.03	23	20	11	10

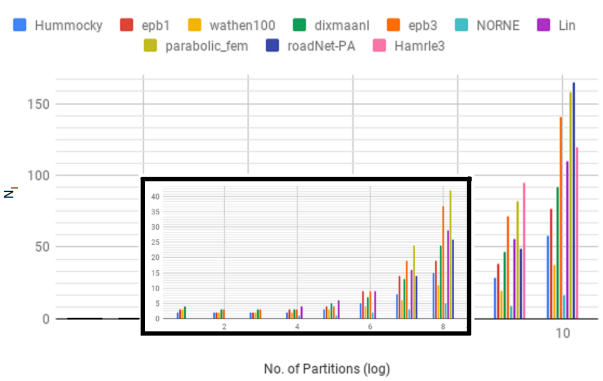
Notice that building $\vec{x_p}$'s is a very time consuming step on the ARM processor (T_{Ba}) whereas it is almost negligible on the I7 (T_{Bi}). The ARM processor is 6.67×10^8Hz while I7 has the slightly higher base frequency of 1.8×10^9. The disproportional increase is most likely due to the high number of memory accesses needed to build $\vec{x_p}$, each time requiring memory access to the DRAM via the high latency AXI bus. The I7 case will thus also be presented as a theoretical case alongside the ARM, since large scale systems will probably not have a microprocessor communicating with the accelerator.

Recall from Figure 7.1 that there are two reference cases against which speedup is achieved, namely *path 1* where the SpMV computed on I7 and *path 2* where SpMV is computed with the HLS kernel non-partitioned. The duration of these paths are referred to as T_{R1} and T_{R2}. Since some benchmarks could not be computed non-partitioned, the derived execution times for 2 partitions from Figure 7.7b are used instead.

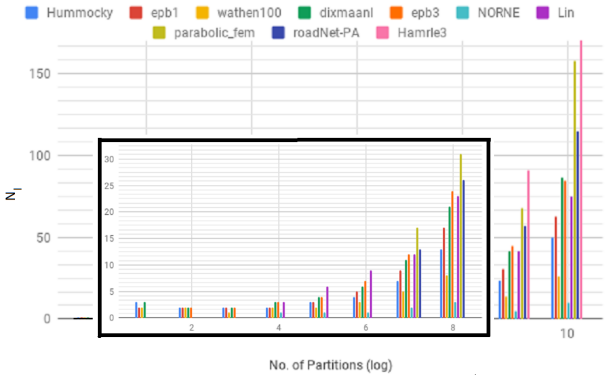
The *sparstition* time (T_S) includes the compression of the `Index Maps` and is **only performed once** whereas T_{Bx} (where x = i or a), and T_{PS} are performed until speedup is achieved, or N_I times. To compute the N_I against reference time T_{Ry} (where y = 1 or 2), we compute how many times T_{Ry} must be computed so that the one-time T_S cost is paid off. Each time we perform T_{Ry} we also perform the partitioned SpMV which consists of T_{Bx} and T_{PS}. The formula is derived in Equation 7.1.

$$\begin{aligned} T_{Ry} \times N_I &= T_S + T_{tx} \times N_I \\ \text{where } T_{tx} &= T_{Bx} + T_{PS} \\ N_I &= \frac{T_s}{T_{Ry} - T_{tx}} \end{aligned} \tag{7.1}$$

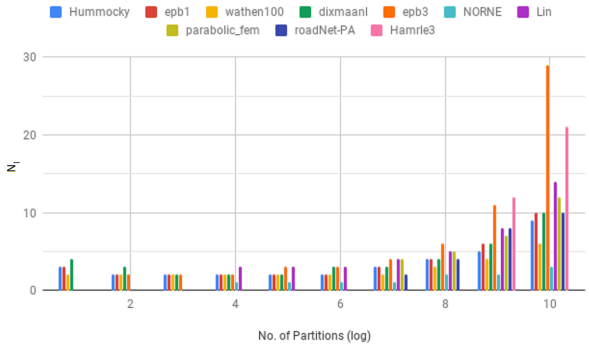
From this equation it is clear that speedup can **never** be achieved when the time it takes to build $\vec{x_p}$ and perform the partitioned SpMV (T_{tx}) is **larger** than the reference time (T_{Rx}. These scenarios are marked with red (comparing with both T_{R1} and T_{R2}) and pink (comparing only with T_{R2}) **x** in the table. Figure 7.8 depicts the number of iterations until speedup is achieved for most benchmarks.



(a) `Index Maps` to -1



(b) Boolean maps



(c) Bitmaps

Figure 7.8: N_I relative to non-partitioned HLS (T_{R1}) for all implementations.

The margin is very narrow for N_p in the range from 2^1 to 2^4, in fact it is so narrow that it does

not warrant further analysis of these implementations. Therefore, the focus will only rest on the bitmap implementation for the remainder of this chapter as it is the most scalable and efficient one.

Finally, we present the minimum N_I for speedup when compared with software execution on I7. We saw in Tables 7.3 and 7.7 that the software performance of I7 beats the non-partitioned HLS for all benchmarks, so naturally we expect larger N_I's. The results are summarized in Figure 7.9.

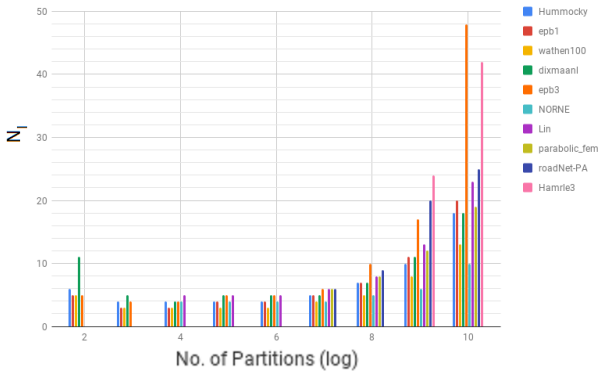


Figure 7.9: Bar chart of N_I needed surpass the I7 execution times for the bitmap implementation.

Notice that for 2 (2^1) partitions, none of the benchmarks are able surpass the I7. That is, the execution time of heavier partition is still longer than that of running the benchmark non-partitioned on the I7. Similar to Figure 7.8c, we note that the N_I stays relatively constant for the first few N_P but then starts to rise rapidly. This is in tune with the increase in the cost of initializing and compressing `Index Map`'s, which increases proportionally with N_P. Since N_P doubles each interval, so too does the overhead.

7.4.2.3 Theoretical Speedup of Bi-CGSTAB

To give an idea on how many iterations are needed to solve a linear system represented by each benchmark, we use Bi-CGSTAB as an example solver with ILU(0) pre-conditioner, and gather the number of iterations it needs to solve our benchmarks. Out of 10 benchmarks, only 8 can be solved. This is because the two largest benchmarks have many zero-values on the diagonal, which is not compatible with our pre-conditioner. The iteration counts are summarized in Table 7.9. Note that the iteration count can vary significantly depending on the initial guess and the vector being solved for. These values may be smaller or larger using other vectors.

Table 7.9: N_I obtained from running Bi-CGSTAB on each of the benchmarks for a single test case.

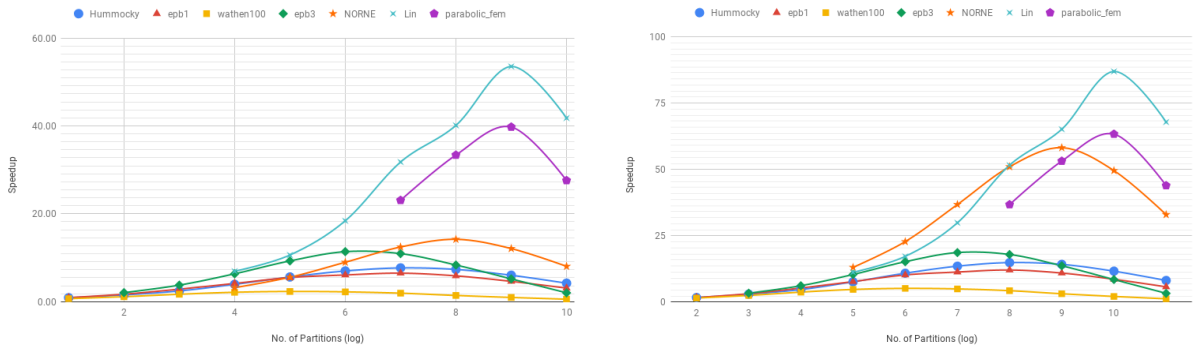
Matrix	Hummocky	epb1	wathen100	dixmaanl	epb3	NORNE	Lin	parabolic_fem	roadNet-PA	Hamrle3
Iterations (N_{I^*})	98	74	7	12	98	84	1419	830	N/A	N/A

These reference numbers are used to calculate the theoretical speedup in a cluster environment shown for both non-partitioned SpMV and I7 (T_{R1} and T_{R2}) in Figure 7.10. The speedup is

calculated with the following formula:

$$speedup = \frac{T_{Ry} \times N_{I^*}}{T_S + (T_{Bi} + T_{PS}) \times N_{I^*}} \tag{7.2}$$

where T_y is the reference time for I7 (y = 1) or non-partitioned SpMV (y = 2), N_{I^*} is the number of iterations obtained from Table 7.9, T_S is the *sparstition* time, T_{Bi} the build time on I7 and T_{PS} the execution time of the longest partition. Note that Bi-CGSTAB and other CG (conjugate gradient) variants perform two SpMV per iteration but the speedup numbers are obtained by considering only one per iteration. Figure 7.10 shows the speedup when N_{I^*} iterations are executed for their respective benchmarks with a varying N_P, both relative to I7 (7.10a) and non-partitioned SpMV (7.10b).



(a) Speedup relative to (non-partitioned) SpMV in software on I7.

(b) Speedup relative to SpMV on HLS kernel without partitioning.

Figure 7.10: Theoretical speedups against non-partitioned HLS kernel and software execution on I7 with N_{I^*} iterations.

The speedup gained is understandably proportional to N_{I^*} as observed with Wathen100 resulting in the least speedup while Lin the most. For all benchmarks there is a *sweet point* where the maximum speedup is achieved followed by a drop. This is the point where the proportionally increasing cost of processing `Index Maps` begins to outweigh the benefit of partitioning, and thus parallelizing, the problem further. Recall from Section 3.2.2 that the processing of `Index Maps` is parallelizable as each `Index Map` is independent from one another. Therefore the peak of the curve can be extended along the x-axis if the task of *sparstition* is distributed over multiple machines.

7.4.3 Peak Performance

Now that we have found the N_I required to gain speedup and computed theoretical speedup for Bi-CGSTAB with N_{I^*}, we will now explore how the speedup scales. We do this by calculating

the average time it takes to perform partitioned SpMV, with the *sparstition* (T_S) cost factored in.

$$P_{its} = \frac{FLOPS}{T_{avg}}$$
$$\text{where } T_{avg} = \frac{T_S + N_{its} \times T_{PS}}{N_{its}} = \frac{T_S}{N_{its}} + T_{PS} \tag{7.3}$$

Since T_{PS}, the combined time of building $\vec{x_p}$ and performing partitioned SpMV, is constant, T_S decreases by a factor of the number of iterations (N_{its}). Let us say that $N_{its} = 2$, that it takes 1ms to do *sparstition* and performing partitioned SpMV (including building of $\vec{x_p}$) takes 0.20ms. Then the *average* time it took to perform partitioned SpMV is $\frac{1+0.20}{2} = 0.60ms$. Table 7.10 presents the theoretical results obtained from running 100 and 2000 iterations.

Table 7.10: Theoretical SpMV performance and speedup after 100 and 2000 iterations for all benchmarks. The performance used as the HLS reference for starred (*) matrices was derived using the formula in Section 7.4.1.

	P_{100} (GFLOPS)										P_{2000} (GFLOPS)										$N_p = 1$ * Performance (GFLOPS)		Maximum Speedup with $N_p = 1$ *			
No. of Partitions / Benchmark	2	4	8	16	32	64	128	256	512	1024	2	4	8	16	32	64	128	256	512	1024			100		2000	
																					HLS	I7	HLS	I7	HLS	I7
Hummocky	0.51	0.87	1.37	2.21	3.15	3.92	4.31	4.13	3.39	2.38	0.52	0.90	1.45	2.44	3.71	5.04	6.18	6.87	7.02	6.91	0.287	0.557	14.83	4.28	24.15	12.41
epb1	0.48	0.89	1.50	2.21	2.97	3.35	3.68	3.43	2.80	1.97	0.49	0.92	1.59	2.42	3.42	4.11	5.05	5.35	5.41	5.43	0.274	0.514	13.44	3.83	19.85	10.57
wathen100	0.52	1.01	1.94	3.60	6.23	9.82	12.87	15.53	17.63	16.74	0.52	1.01	1.94	3.60	6.23	9.82	12.87	15.53	17.63	16.74	0.266	0.593	28.31	6.72	64.98	28.22
dixmaanl			1.27	2.18	3.41	4.52	5.03	4.55	3.67	2.54			1.36	2.46	4.26	6.52	8.61	9.75	10.22	10.78	0.281	0.526	16.83	4.83	36.05	20.47
epb3		0.31	0.58	1.12	2.01	3.36	4.81	5.88	7.27	5.10		0.31	0.59	1.13	2.06	3.51	5.25	7.05	11.52	18.76	0.310	0.509	21.32	10.01	55.03	36.81
NORNE*				1.91	3.37	5.56	7.96	9.37	8.18	5.53				2.09	4.02	7.71	14.39	24.84	34.72	39.14	0.143	0.586	65.50	9.44	273.62	66.74
Lin*				3.04	4.27	5.89	6.75	5.83	4.15	2.40				3.83	5.88	10.28	18.09	23.47	33.96	28.28	0.338	0.549	19.98	4.38	100.44	51.47
parabolic_fem*							5.98	5.78	4.52	2.74							13.90	22.15	31.43	23.73	0.344	0.548	17.38	4.99	91.35	43.27
roadNet-PA*							4.18	3.21	1.48	1.15							13.23	18.85	17.52	15.63	0.112	0.291	37.23	3.94	167.83	53.72
Hamrle3*									2.64	2.05									31.33	27.96	0.066	0.132	39.98	15.46	474.05	211.03

We repeat that running a high number (> 32) of partitions in parallel is unrealistic due to the amount of hardware and bandwidth needed. Also 2000 is a high number of iterations but it demonstrates the saturation point for the smaller matrices. Generally, as N_P increases so does the performance. This is due to T_S gradually becoming a non-factor and thus leaving the execution time of the longest SpMV partition as the only performance factor.

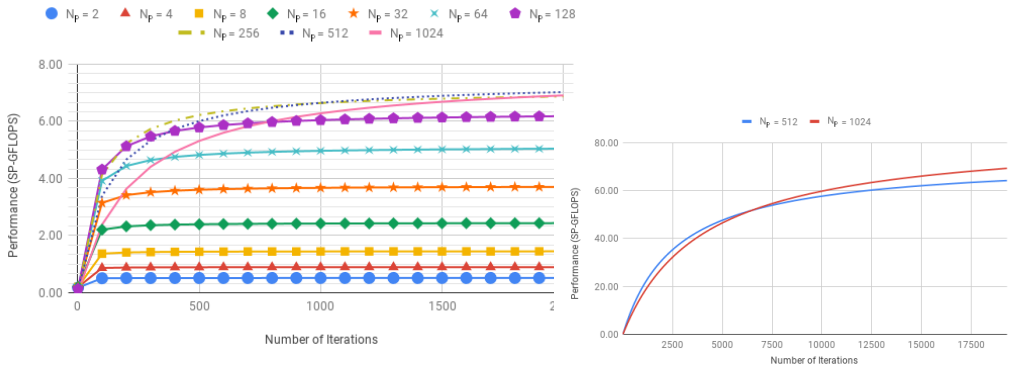
There is significant speedup for all benchmarks after 100 iterations which is a promising sign. Running more iteration does not affect the performance for a low number of partitions as the relatively low *sparstition* cost is quickly covered by the faster parallel execution times. It naturally takes the larger matrices many more iterations to saturate.

We note in Figure 7.11a that a performance *saturation point* (SatP), i.e. the point where T_S becomes a non-factor, is reached much quicker for smaller N_P due to smaller T_S cost. The reason we sometimes observe lower performance for $N_P = 1024$ compared with $N_P = 512$ is that it takes more than 2000 iterations to reach the SatP. We observe in Figure 7.11b that there is eventually an intersection point.

Naturally, it takes more iterations to saturate Hamrle3 in Figure 7.11 due to larger T_S. However, once the SatP is reached, the performance increase is much higher than that of the smaller matrices. It is, on the other hand, quite unlikely to ever be saturated in practice as it requires almost 20000 iterations. Not to mention the cost of 512 to 1024 accelerators.

7.4.4 Comparison with State-of-the-Art

We are now ready to compare our results with two state-of-the-art designs. Table 7.11 presents our next to [2] and [11].



(a) The scalability of Hummocky performance with an increasing number of iterations.

(b) The scalability of Hamrle3 performance with an increasing number of iterations.

Figure 7.11: The performance scalability of performing multiple partitioned SpMVfor two benchmarks.

Table 7.11: This work compared with the state-of-the-art SpMV kernels on FPGAs. Our results are obtained from running 100 iterations.

Works	Device	Benchmark			Performance (GFLOPs)		Bandwidth (GB/s)	
		Name	Dim	Nnz (%)	Single	Double	Single	Double
[2]	Stratix V	raefsky1	3242	2.80		3.99		38.4
		epb1	14734	0.044		0.69		19.2
		scircuit	170998	0.0033		0.08		19.2
[11]	Convey HC-2x	raefsky1	3242	2.80		1.83		80
		epb1	14734	0.044		1.03		
		scircuit	170998	0.0033		0.792		
This work (estimated)	ZYNQ-7000	Hummocky	12380	0.08	3.15*	1.58**	64*	64*
		epb1	14734	0.044	2.97*	1.49**		
		Lin	256000	0.003	4.27*	2.14**		

We obtain the starred (*) results with the Equation 7.3 from the previous section for 100 iterations. We then choose 32 partitions and emulate an environment where each one is executed in isolation. We choose 32 partitions because each HLS kernel is assumed to have bandwidth of 2GB/s as was the case for our design. The total bandwidth of the system theb becomes 64GB/s which is between the 80G/s reported by [11] and the maximum of 38.4 GB/s reported by [2]. However, all of our results are single-precision so we make the assumption that halving the performance is identical to the double precision performance (**).

The only shared benchmark is epb1, which our design might scale to outperform. The other benchmarks are unfortunately not included in our set.

7.5 Load Balancing

7.5.1 Verification of the Weights Model

We defined a formula to predict the execution time of the partitions during run-time of *sparstition* in Section 4.3. To put it to the test, we varied the partitioning point for NORNE. Therefore N_P is equal to 2 and we measured the effect it had on each partition. Then we plotted the weight formula against the results and adjusted the curves. Recall that the weight formula is meant to give *relative* execution times. The result is summarized in Figure 7.12.

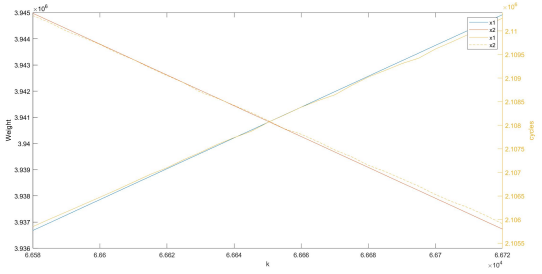


Figure 7.12: The yellow curves are execution times in cycles and correspond with the right axis. The blue and red curves are the weight formulas and correspond with the left axis.

As we can see, the weights formula fits very well with the actual execution times.

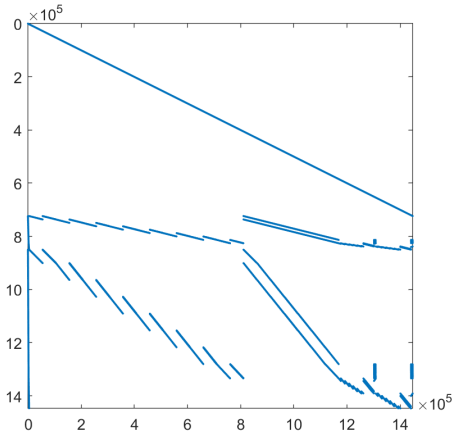
7.5.2 Load Balancing the Execution of Multiple Pipelines

So far we have only explored the case of a single pipeline as the isolated execution did not use the pipelined functionality of the kernel. There is no upper limit for N_P, but as it rises it becomes increasingly difficult to execute each partition in isolation due to the need for a dedicated accelerator. That is, the number of accelerators (N_A) does not scale in practical systems. The subsequent discussion focuses on the case where $N_A < N_P$ which requires us to group partitions together and assign said groups to the smaller number of accelerators. In the example that follows, Hamrle3 depicted in Figure 7.13a, is split into 1024 partitions and an environment of 32 accelerators is emulated. Therefore each accelerator is assigned 1024/32 = 32 partitions each which are executed in a pipeline.

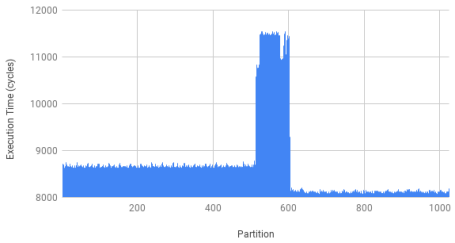
The isolated execution exposes highly irregular execution times resulting from the sparsity pattern, which is characterized by the sparse upper half and dense lower half. For the following example, $N_A = 32$ and the task is to assign these partitions to the available accelerators. The most simple approach is to simply group adjacent partitions together, which results in the assignment depicted in Figure 7.14a.

However, this results in significant variance between the groups. Another approach is to split the partitions into three major groups characterized by the length of execution time. Each major group is further divided into 32 sub-groups, one for each accelerator as shown in Figure 7.14b. Finally, each accelerator is assigned one sub-group from every major group. The resulting execution times of these two schemes is presented in Figure 7.15.

The extremes are noticeably filtered out, but the change in execution times is not significant in this case due to the small accelerator cache and the efficient pipelining. This is a potential

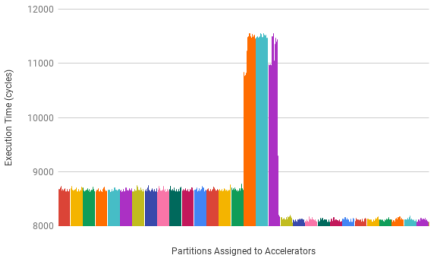


(a) The sparsity pattern of Hamrle3.

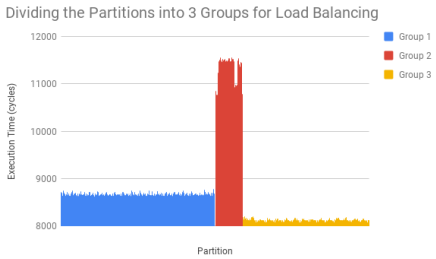


(b) The execution times Hamrle3 partitions for N_P = 1024.

Figure 7.13: Figure comparing the sparsity pattern of Hamrle and the uneven execution times for N_P = 1024.



(a) The 1024 partitions assigned to 32 groups where each color corresponds to an accelerator.



(b) The 3 major groups identified for even distribution.

Figure 7.14: The two distribution schemes, grouping adjacent partitions together versus identifying groups and subdividing them.

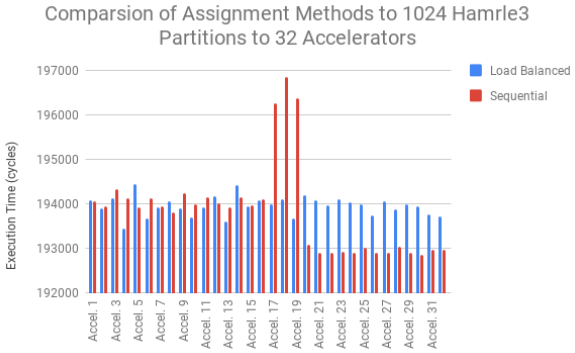


Figure 7.15: Load balanced execution times versus normal execution times for each accelerator

topic for future research where the platforms have larger caches and load balancing becomes more vital. The identification of the three groups was done manually for this example but some advanced technologies such as machine learning might assist with this task in the future.

7.6 Conclusion

We can now conclude that the co-design presented in this thesis shows promise once all costs have been factored in. The design works as intended, which enabled us to compute SpMV with benchmarks that have $\vec{x}$ that are up to 34 times larger than the available cache. The kernel by itself proved to be relatively efficient despite many shortcomings of HLS by utilizing as much of the available bandwidth as possible. It was compared with the current state-of-the-art which also tested its capability of computing double precision floating points. Additionally it achieved up to 150x speedup against the ARM microprocessor while performing slightly worse against Intel I7 processor.

We evaluated the cost of *sparstition* and its three implementations in detail. We eventually found out that due to the proportional increase in cost of processing `Index Maps` that using bitmaps was the most efficient due to its scalability. However, if the *sparstition* is split and computed in parallel on multiple machine, the other implementations are strong candidates.

The *sparstition* and partitioned SpMV were computed together in order to find N_I presented in Figure 7.1 at the beginning of the chapter. The numbers proved to be reasonable when compared with the computation of Bi-CGSTAB solver preconditioned with ILU(0). The theoretical speedups in iterative algorithms reached 70x when compared with SpMV without partitioning on the kernel, and 50x compared with the software execution on I7. We also explored how the speedup scales in terms of performance in FLOPS as the *sparstition* cost is gradually factored out. Every benchmark was evaluated with a special attention given to the smallest (Hummocky) and the largest (Hamrle3). We saw that performance *saturates* as the number of iterations increases. Finally, we explored the case when the number of partitions exceeds the number of accelerators. We accomplished this by emulating an environment with multiple pipelined HLS kernels, and showcased that the platform can perform (coarse-grained) scheduling with ease.

In the following chapter, we will conclude this work with a brief summary of what has been discussed. The work that may ensue in the future is also listed.

8 Conclusion and Future Work

In the previous chapter, we evaluated every major aspect of our design, the *sparstition* algorithm, the HLS kernel and their interplay. We conclude our work in this thesis with a brief summary of the key points starting from the problem statement formulated in the introduction chapter. Subsequently, we state the contributions from this thesis and highlight future work.

8.1 Summary

We have now presented a solution to the problem that we introduced in the first chapter of this thesis. We identified the problem that HLS kernels were not performing on par with their manually designed counterparts, mainly due to difficulties in managing multiple parallel pipelines. Instead of suggesting improvements to the HLS compiler itself, we approached the problem from a higher level and formulated the following problem statement:

Can a partitioning algorithm be developed which splits SpMV into multiple independent streams, and show promise in achieving speedup in HLS designs within a reasonable amount of iterations?

So essentially we wanted to find out, whether it is possible to efficiently partition SpMV in such a way that multiple pipelines running on separate computational units can execute in parallel. In order for the solution to be efficient, the cost of performing the partitioning should be quickly be repaid so that the benefits of the parallel execution can be reaped.

In order to address this problem statement, we first designed an HLS kernel that was efficient in computing multiple partitions by overlapping communication with computation. We evaluated the performance of the kernel as a standalone unit to obtain reference times to compare against. We also compared with current state-of-the-art design using the benchmarks presented in the respective work. Our results achieved speedup, even though the other work was performed in simulation. For the larger benchmarks that we could not execute non-partitioned, we derived the estimated performance if two partitions were executed in a pipeline. We found this to be reasonably close to non-partitioned execution times for the smaller benchmarks. We also computed a naive implementation of SpMV in software using a state-of-the-art processor.

Then we evaluated the `Sparstitioner` , which consists of initializing and compressing `Index Maps` and the *sparstition* algorithm. We created three different implementations and after extensive evaluation, it was concluded that bitmaps scaled the best. Our findings were then used to finally compute theoretical number of iterations required until speedup is achieved (N_I) in Section 7.4.2.2, which proved reasonable compared with reference runs of the Bi-CGSTAB solver, summarized in Table 7.9. The N_I was demonstrated to depend on the time it takes to perform the pre-processing, and the speedup gained from the parallel execution of SpMV .

As an extra step, we investigated how the performance scaled with 2000 iterations, which is a significant number but the shows that the platform scales well for larger benchmarks in

the future. However, we found out that the performance reached a *saturation point* which is where the cost of *sparstition* was fully covered by the speedup of parallel SpMV. This occurred faster for smaller benchmarks as the pre-processing time was much smaller for them as seen in Figure 7.3.

The problem statement was successfully addressed but the work is far from complete. We only showed our results in emulated environments and we are yet to prove that the speedup is realizable in practice. The next section presents the contributions made by this thesis.

8.2 Contributions

The contributions of this work are the following.

- The design and implementation of an HLS kernel that improves the state-of-the-art HLS SpMV designs. We achieved this by making use of as much bandwidth as possible within the ZYNQ platform. In addition, the kernel features an efficient pipeline so that communication can be transferred with computation when multiple SpMV partitions are computed.
- An efficient and parameterized partitioning algorithm that does not require expensive post-processing. The algorithm worked well with the HLS kernel and can, in theory, be used to improve any implementation.

8.3 Future Work

The work presented in this thesis is far from being complete as only a proof-of-concept was presented and evaluated. The following is a non-exhaustive list of work that may ensue.

Handling Dense Rows: If a matrix contains a dense row, which has more non-zero columns than can fit into the available memory specified by the user, the algorithm cannot find a solution and exits. It is unclear how this problem can be approached, as one of the characteristics of the *sparstition* is that no reduction is required. But unfortunately, that is not possible with all sparsity patterns.

Parallelizing the Sparstitioner: We exposed that the `Sparstitioner` is highly parallelizable in Figure 3.5 as each partition can be processed independently. The performance would improve substantially as a result, because the workload of the two largest bottlenecks, namely *sparstition* and the `Index Maps` processing, would be distributed over different computational units. The implementations that were promising for a small number of partitions, namely Boolean maps and setting `Index Maps` to -1, might turn out to be more scalable.

Executing SpMV in parallel: We only emulated environments that had multiple computational units in this work. A logical next step is therefore to actually perform the SpMV in parallel with the *sparstitioned* partitions. As we noted, the design of *sparstition* is *implementation agnostic* and thus we are not limited to the HLS kernel. We could, for example, continue with GPUs (Graphical Processing Unit) or multi-core CPUs (Central Processing Unit).

Speeding Up Solver Algorithms: We obtained reference numbers from running Bi-CGSTAB to verify that the number of iterations until speedup was reasonable for our benchmarks. Indeed, we obtained theoretical speedup for every benchmark that converged thus demonstrating the promise of this work. However, there are significant challenges in integrating our

solution to the solver algorithm which must be addressed. The step before SpMV in Bi-CGSTAB is the pre-conditioning which is usually highly sequential. To maximize the benefits of SpMV , the streaming to the computational units would need to begin as the pre-conditioned vector becomes available. Therefore, FPGAs show the most promise in running the host-code as they are capable of achieving task-level pipelines. The next challenge is receiving the result from the cluster, which could be relatively simple if a dot product is to be computed as the operations are associative. However, perhaps multiple dot-products need to be computed following SpMV which are also applied with the pre-conditioner, as is the case with the second SpMV of the Bi-CGSTAB method.

Porting the Solver to a Cluster: If the solver algorithm discussed previously shows promise and significant speedup is achieved, then we can look into porting the solution to a cluster as we illustrated in Figure 3.8. There are significant challenges involved in streaming data to and from the worker nodes so the the implementation must be made as efficient as possible. Preferably, the solver runs on an FPGA so that the communication overhead is masked by the computation.

Bibliography

[1] Xilinx, Inc., *Vivado HLS Optimization Methodology Guide*, 2017.

[2] P. Grigoras, P. Burovskiy, and W. Luk, "Cask: Open-source custom architectures for sparse kernels," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 179–184. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847338

[3] R. Garibotti, B. Reagen, Y. S. Shao, G. Wei, and D. Brooks, "Assisting high-level synthesis improve spmv benchmark through dynamic dependence analysis," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1440–1444, Oct 2018.

[4] "Altera gives intel a hot hand in programmable chips." [Online]. Available: http://fortune.com/2015/12/28/intel-completes-altera-acquisition/

[5] "Ibm and nvidia present the nvlink server you've been waiting for," Oct 2018. [Online]. Available: https://www.ibm.com/blogs/systems/ibm-nvidia-present-nvlink-server-youve-waiting/

[6] H. Liu, L. Shen, Y. Chen, K. Wang, B. Yang, and Z. Chen, "A parallel simulator for massive reservoir models utilizing distributed-memory parallel systems," *CoRR*, vol. abs/1701.06254, 2017. [Online]. Available: http://arxiv.org/abs/1701.06254

[7] U. M. Ascher and E. Boxerman, "On the modified conjugate gradient method in cloth simulation," *Vis. Comput.*, vol. 19, no. 7-8, pp. 526–531, Dec. 2003. [Online]. Available: http://dx.doi.org/10.1007/s00371-003-0220-4

[8] S. Skalicky, C. Wood, M. Lukowiak, and M. Ryan, "High level synthesis: Where are we? a case study on matrix multiplication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013, pp. 1–7.

[9] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 78–85.

[10] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174264

[11] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 3:1–3:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555729.2555732

[12] K. Townsend and J. Zambreno, "Reduce, reuse, recycle (r3): A design methodology for sparse matrix vector multiplication on reconfigurable platforms," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, June 2013, pp. 185–191.

[13] M. Taouil, "A hardware accelerator for the openfoam sparse matrix-vector product," 2009, uuid:ce583533-45ea-4237-b18d-fe31272ea1ee.

[14] J. Pinhao, "Fpga multi-processor for sparse matrix applications," 2015.

[15] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, 2003, aAI3121741.

[16] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[17] Xilinx, Inc., *AXI Reference Guide - UG761*, 2011.

[18] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 36–43.

[19] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046202

[20] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554785

[21] S. Cheng and J. Wawrzynek, "Architectural synthesis of computational pipelines with decoupled memory access," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 83–90.

[22] S. T. Fleming and D. B. Thomas, "Using runahead execution to hide memory latency in high level synthesis," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 109–116.

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[24] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, "Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods,"

SIAM Journal on Scientific Computing, vol. 37, no. 5, pp. S602–S626, 2015. [Online]. Available: https://doi.org/10.1137/140980260

[25] J. Pinhao, W. Jose, H. Neto, and M. Vestias, "Sparse matrix multiplication on a reconfigurable many-core architecture," in *2015 Euromicro Conference on Digital System Design*, Aug 2015, pp. 330–336.

[26] "The university of florida sparse matrix collection." [Online]. Available: www.sparse.tamu.edu/

List of definitions

.. ...

Sparsity Pattern of Benchmarks