Dataflow Hardware Design for Big Data Acceleration Using Typed Interfaces

by

Ákos Hadnagy

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday August 26, 2020 at 13:00.

Student number:	4821106	
Thesis committee:	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. ir. T. G. R. M. van Leuken,	TU Delft
	Ir. J.W. Peltenburg,	TU Delft

Abstract

Recent trends in large-scale computing demonstrate continuous growth in the need for raw processing performance. At the same time, the slowdown of vertical scaling pushes the industry towards more energy-efficient heterogeneous architectures. With the appearance of FPGAs in the cloud and data centers, a new architecture is offered for offloading processing tasks and to bundle custom processing hardware with the applications. However, with great adaptability comes the increased complexity of development. The adoption of custom accelerators has been bounded by their limited programming models and the long turnaround time of development.

In this thesis, we look at current trends in the digital hardware design and synthesis to evaluate them in a big data context and identify the bottlenecks that limit productivity in the development and integration of domain-specific accelerators.

Based on the findings, we propose a composition language for components that implement typed interfaces to streamline kernel development. The language allows developers to compose accelerators from individual processing units that implement custom dataflow interfaces in a productive way. The productivity boost and utility of the language were evaluated on a practical use-case, showing almost two orders of magnitude reduction in code size. The performance of the proposed approach was benchmarked on a Power9 system with OpenCAPI, where our proof-of-concept accelerator kernel was able to achieve $4.04GB/s$ throughput using only 3.75% of the FPGA resources. The integration of the accelerator led to a $13x$ speedup compared to a CPU-based Apache Spark implementation of the same algorithm.

Preface

This thesis is the result of a nine-month effort to conduct novel research in big data acceleration at the Accelerated Big Data Systems group. For me, personally, it has been a tremendous learning experience. These nine months gave me the opportunity to dive into many technologies I wouldn't have encountered anyway. Although I was following the field before the start of this thesis, I was blown away by the vibrant research that's being carried out. The development moves so fast that solutions appeared and got discontinued during the course of this thesis. I can safely say that we're witnessing large-scale acceleration becoming a hot-topic, it's simply the best and most rewarding time to be involved.

I am grateful for being a part of the ABS group for the last nine months; I met many knowledgeable and passionate people there. The level of research and engineering ingenuity present in the group always astonishes me. Special thanks to Joost, Johan, and Matthijs from the group for their support and valuable ideas.

I would particularly like to thank Dr. Zaid Al-Ars, my supervisor, whose energy and drive never ceased to amaze me. His door was always open, even for lengthy discussions, so no stones were left unturned. This thesis was written during troubled times in the world, making Zaid's positivity and excitement about all the small steps ever so motivating.

I also want to thank Stefan Hofman for giving constructive feedback on the report, and for the (too?) many coffee breaks we shared.

Finally, I must express my very profound gratitude to my parents *Ilona* and *Attila* for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

Ákos Hadnagy
Kalocsa, 10th August 2020

Contents

List of Figures

1 Introduction

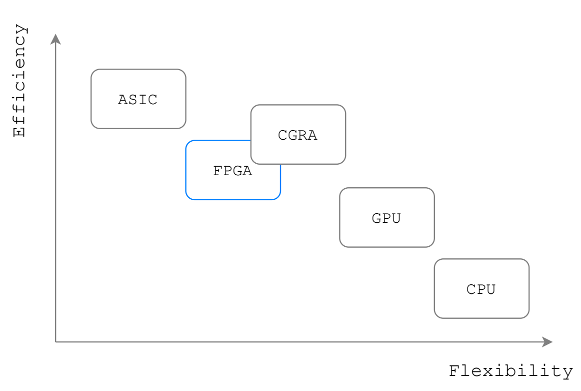
1.1. Context

The data generated by mankind is growing at an exponential rate. Mining this data presents invaluable opportunities throughout a wide span of industries, such as increasing customer satisfaction, predicting trends, extracting information for the improvement of processes, or by finding new ways to develop medicine, and optimizing healthcare.

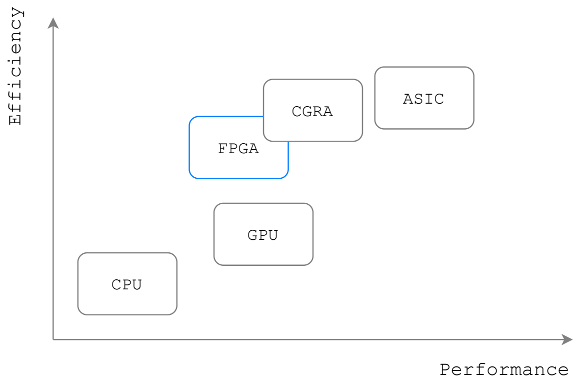
With the increase in data volume and velocity, a point has been reached where more data is being produced than what is possible to store and flows in faster than it is possible to process. As a result, the need for large-scale computing platforms has significantly intensified.

As a consequence to the slowdown of Moore's law, the failure of Dennard-scaling, and the emerging problem of dark silicon, the development priority for general-purpose computing fabric is shifting from performance to energy efficiency to counteract the tighter power budgets. To achieve higher energy efficiency, more specialized hardware is required as shown in Figure 1.1. However, specialized hardware, such as ASICs (application-specific integrated circuit) suffer from higher nonrecurring engineering costs compared to more flexible architectures due to their shorter life-cycle and tighter target application domain. Therefore, for modern computing architectures, both energy-efficiency and flexibility are requirements. Reconfigurable hardware architectures, such as FPGAs (field-programmable gate array) and CGRAs (coarse-grain reconfigurable architecture) appear to be promising to balance flexibility, energy-efficiency, and performance.

With the appearance of FPGAs in the datacenters and cloud [14], distributed application developers are presented with the powerful opportunity of being able to bundle custom processing hardware with their application. A significant portion of big data workloads consists of a few, massively parallel workloads that can be executed in a large scale, and are suitable for more efficient implementations by utilizing custom hardware. FPGAs are being deployed for various compute-intensive big data applications, such as genomics algorithms [21, 30], data decompression [17], and image processing [20]. However, these applications are routinely written in high-level distributed frameworks, which are notoriously far from the bare-bone hardware. Accelerating these applications requires to bridge a vast semantic gap, involving a thick stack of tools and engineers with a wide area of expertise.



(a) Efficiency vs. flexibility.



(b) Efficiency vs. performance.

Figure 1.1: Architecture comparison regarding flexibility, performance, and efficiency. [28]

Challenges

A survey of hardware acceleration on a datacenter scale was published in 2016 by S. Yesil et al. [41] that categorizes the open problems in the field:

- Host-accelerator, accelerator-accelerator interface
- Memory hierarchy
- Programming and management of accelerators

These can still be viewed as challenges in 2020; however, significant progress can be seen in all of the areas mentioned. For example, OpenCAPI offers high bandwidth coherent access to system memory for accelerators, while HBM is making its way onto the FPGA accelerators, improving the memory hierarchy. On the management side, Blaze [22] is a promising solution to support datacenter-scale deployment of FPGA accelerators.

Extensive and broad research is being carried out on programming custom accelerators, but to this day, it remains a deterrent force to the wide adoption because of the limited programming models and missing straightforward abstraction layers between software and gate-level hardware. While the hardware offerings and adoptions are widening, developing heterogeneous analytics applications with a low turnaround time remains a bottleneck.

1.2. Problem statement

This thesis aims at addressing the accelerator kernel programming, development and generation problem. Based on the challenges discussed above, we can formulate the following research questions:

- What are the current trends in digital hardware design which promise productivity improvements?
- Which solutions are relevant to be used in a big data context?
- Which solutions would be most suitable to be used in a workflow with Apache Arrow and Fletcher?

To carry out the research, we employed an experiment- and discovery-oriented approach by evaluating a broad range of solutions in the context of the target application domain and applying the findings to build a new proof-of-concept workflow.

Thesis contributions

The contributions of this thesis can be summarized as follows:

- **Study of solutions for kernel development** — A wide-span outlook of technologies has been presented for kernel development in a big data context and analyzed in terms of strengths, weaknesses and applicability. The surveyed solutions range from modern HDLs to complete end-to-end frameworks for acceleration.

- **Chisel backend for Tydi** — A Chisel backend has been developed for the Tydi reference implementation to support interface and kernel skeleton generation in a modern HCL (Hardware Composition Language).
- **Graphviz backend for Tydi** — A Graphviz backend has been developed for the Tydi reference implementation to be able to generate visualizations for structural implementations.
- **Composition language prototype for Tydi** — A prototype language specification has been proposed for the purpose of building structural designs using components that implement Tydi interfaces. The language also proposes a number of constructs that move complexity related to handling complex types and dimensionality from the components to the composition language. The specification is provided as a PEST parser PEG (parsing expression grammar), in addition, AST (abstract syntax tree) transformations have been implemented for the constructs featured in the proof of concept design.
- **Proof of concept design** — A proof of concept accelerator has been built using the FilterStream and ReduceStream patterns to show the potentials in the target application domain. The design has been tested and profiled on a Power9 system with OpenCAPI.

1.3. Outline

The thesis report is organized as follows:

- **Chapter 2: Background** introduces the relevant background knowledge and technologies.
- **Chapter 3: Solution architecture** presents the analysis and comparison of existing technologies, the identified gaps, and proposes a solution that irons out some of the identified shortcomings.
- **Chapter 4: Prototyping** introduces the proposed language and language constructs along with details about their hardware implementations.
- **Chapter 5: Evaluation & results** describes the results of the implemented proof-of-concept design.
- **Chapter 6: Conclusions and recommendations** summarizes the work and lessons learned in addition to providing future development and research ideas.

2
Background

2.1. Apache Arrow

Apache Arrow is described as “a cross-language development platform for in-memory data” [1]. It includes a specification for in-memory data representation, and also provides computational libraries, zero-copy messaging, and interprocess communication for various programming languages.

Since the specification for the physical memory layout is language-independent, the data structures stored in Arrow can be accessed without serialization/deserialization overhead across components written in different languages.

Apache Arrow is built to be performant on systems that support vectorization. It is a columnar format, meaning that it contiguously stores values belonging to the same column, enabling vectorized processing when executing operations on columns, which is a common access pattern in data analytics. In addition, the columnar format helps to optimize memory accesses when requesting parts of a column.

The advantages of columnar format in data analytics:

- Data locality for sequential accesses.
- Constant time random access.
- Possibility of vectorization.
- Easily relocatable in memory.
- Columnar compression schemes can be leveraged.

In the big data field, data often comes in a tabular format. Arrow proposes the *RecordBatch* abstraction to handle such data efficiently. A RecordBatch bundles several columns that are stored in *arrays* with equal length. Arrays are characterized by the logical data type and consist of a sequence of buffers (contiguous memory region), such as *offset* buffers, *validity* buffers, and *value* buffers. RecordBatches also contain meta-data, called *schema* that describes the structure of the data set, including the types of the fields.

FPGA accelerators can also benefit from this standardized memory representation since they are also subject to the serialization/deserialization bottleneck. In order to leverage the full potential of FPGA accelerators, the bandwidth of the data transfer between the host and the accelerator hardware has to be maximized. This goal can only be achieved if the data resides in contiguous buffers in the memory. In addition to providing a standardized format, Arrow stores the data in a contiguous way whenever it’s possible, making it a favorable solution for feeding streaming dataflow hardware.

Leveraging this standardized and well-engineered format, which is currently interfaced with 11 languages, accelerator hardware can be integrated in an efficient and convenient way with a wide variety of big data frameworks. [32].

The Fletcher [32] framework already offers acceleration integration with Arrow by generating hardware layers for reading and writing Arrow-formatted data, in addition to runtime libraries to interface with software.

2.2. Fletcher

While Apache Arrow aims to be a universal, performance oriented data format to be used across the analytical tools, it is also a good candidate for exchanging data between applications and accelerators.

Fletcher [32] is a framework that provides hardware/software interfaces between the Arrow data structures in memory and custom hardware accelerators.

2.2.1. Development flow

The high-level overview of Fletcher is shown in Figure 2.1. The accelerated application interacts with the accelerator by reading and writing Arrow RecordBatches and through a set of API functions. Based on the Arrow schemas, Fletcher will generate a hardware design with developer friendly dataflow interfaces that can be used to read and write Arrow arrays in memory. The accelerator kernel can be developed in any language as long as it is possible to interact with the custom stream types. Behind the generated interfaces, there are multiple platform-agnostic and platform-specific hardware layers to interact with the host.

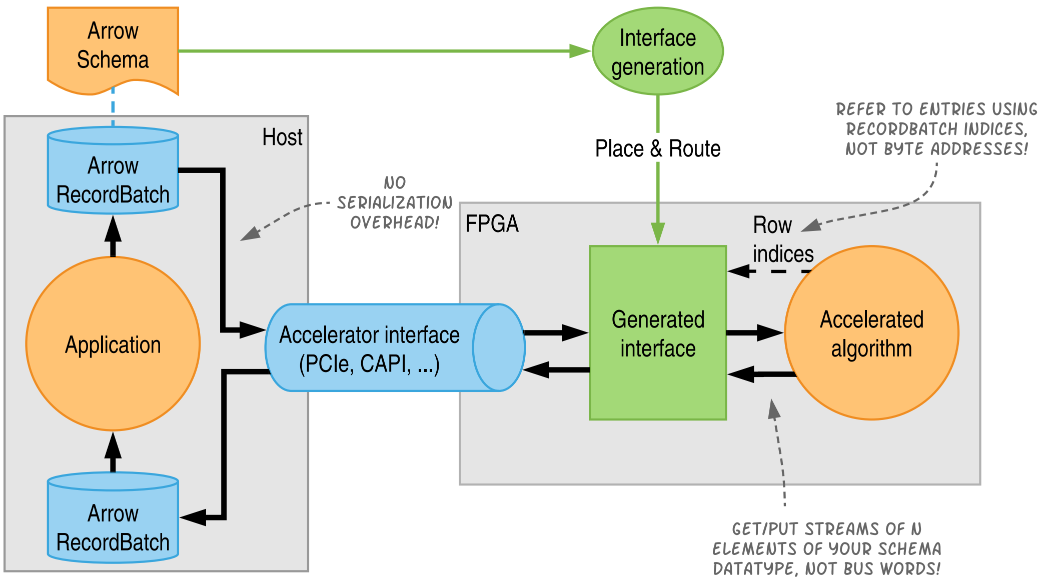

Arrow Schema
Interface generation
Host
Arrow RecordBatch
Place & Route
REFER TO ENTRIES USING RECORDBATCH INDICES, NOT BYTE ADDRESSES!
NO SERIALIZATION OVERHEAD!
FPGA
Row indices
Application
Accelerator interface (PCIe, CAPI, ...)
Generated interface
Accelerated algorithm
Arrow RecordBatch
GET/PUT STREAMS OF N ELEMENTS OF YOUR SCHEMA DATATYPE, NOT BUS WORDS!

Figure 2.1: Fletcher overview. [2]

2.2.2. Hardware design

The schematic overview of the hardware architecture behind the generated interfaces is shown in Figure 2.2. The components present in the hardware architecture are all derived from the Arrow schema. The ColumReaders/Writers are supplied with a configuration string, based on which the internal hierarchy of the component is recursively instantiated to adjust to the specific type of the field in the schema. [31] The ColumReaders/Writers are hierarchically grouped together, each supplied schema results in a RecordBatchReader/Writer. Finally, a wrapper is generated around all the RecordBatchReaders/Writers which also contains the appropriate bus infrastructure and the kernel instantiation.

The accelerator kernel is presented with element streams that are capable of delivering multiple elements per cycle, and command streams, on which the kernel can request a range of items from the memory.

The Fletcher core library and the generated code is hardware-agnostic, the specific platforms are targeted using platform-specific Fletcher runtimes and hardware wrappers. At the time of writing, the following platforms are supported end-to-end or work-in-progress: *Amazon EC2 F1*, *Xilinx Alveo*, *Intel OPAE*, *OpenPOWER CAPI SNAP*, and *OpenPOWER OpenCAPI OC-Accel*. The integration is done using platform-specific wrappers and host-side software libraries.

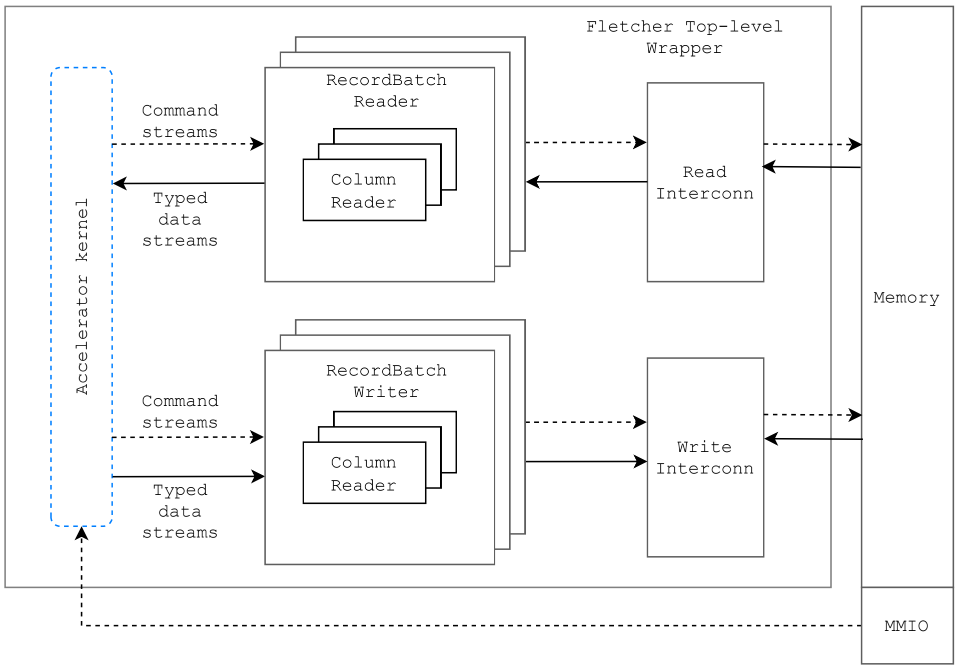


Figure 2.2: Fletcher hardware overview. [32]

2.3. Tydi

Streaming dataflow designs are built from hardware components that are connected using streams in a producer-consumer fashion. The protocol used for data transfer allows for bidirectional synchronization using a handshaking mechanism, Figure 2.3 describes such a protocol. The producer asserts the valid signal when there is a new valid value on the data lines, and the consumer asserts the ready signal when it is ready to consume data. If at the rising edge of the clock, both the valid and the ready signals are asserted, the transfer takes place.

Traditionally, streaming data between components is done using an *AXI4-Stream*-like protocol. However, the AXI4 specifications were originally designed for the standardization of on-chip CPU buses, which limits its flexibility in use-cases where more general components are being used.

For instance, *AXI4-Stream* transfers are byte-oriented, meaning that data can only be exchanged using transactions carrying one or more bytes. This is a reasonable assumption for CPUs, since CPU instructions are traditionally byte-, or word-oriented, similarly to how the system memory is organized. However, in FPGA and ASIC designs, often custom streaming protocols are being used, where this limitation is not necessary or appreciated. Many of the modern hardware description languages support the composition of custom dataflow designs by providing standard libraries with primitives. Examples of these languages are given in Chapter 3.

Streaming dataflow designs have widely settled on streams with *ready*, *valid*, and *data* signals, where the *data* vector has an arbitrary bitwidth, allowing the developers to pack composite data types into a single transfer. However, there is no standard way of mapping complex data types and structures onto hardware streams. This results in incompatibility between components developed in different frameworks or by different engineers. Accelerators often require a number of components to be integrated together, so the developers have to make sure that the interfaces are compliant, or conversion logic has to be inserted.

Tydi (Typed Dataflow Interfaces) [33] proposes a specification for transferring composite and dynamically sized data structures on hardware streams, including multi-dimensional sequences, unions, variants and nested structures.

Tydi proposes a new abstraction to describe the flow of data, called *streamspace*. Streamspace reasons about streams not only in space (e.g. bit-vectors), but also in time: stream transfers. This allows the mapping of complex, dynamically sized data types in a formal way.

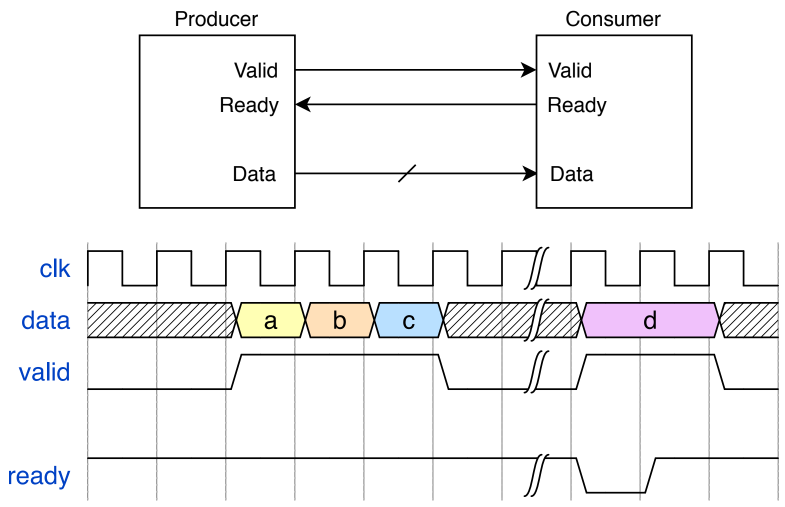


Figure 2.3: Streaming protocol used in dataflow designs.

2.3.1. Type system

The mapping of higher-level data structures is done by defining *logical streams*, which transport a top-level data structure by bundling together multiple physical streams and user-defined signals.

A logical stream node is defined as follows:

$$Stream(T_e, t, d, s, c, r, T_u, x)$$

where:

- T_e - the data type carried by the logical stream. It can include nested $Stream$ nodes.
- t - throughput ratio. It represents the number of elements required to be transferable on the child stream per every element in the parent stream. t is a positive real number.
- d - dimensionality. It represents the dimensionality of the child stream w.r.t. its parent stream.
- s - synchronicity. It represents the relation between the dimensionality information of the parent stream and the child stream. Must be *Sync*, *Flatten*, *DeSync*, or *FlatDesync*.
- c - complexity level.
- r - direction. It represents the direction of the stream w.r.t. its parent stream. Must be *Forward* or *Reverse*.
- T_u - logical stream type representing the *user* signals. Only element-manipulating types are allowed.
- T_u - boolean, representing whether the stream carries information beyond the *user* and *data* payloads.

Tydi defines 6 basic types that can be used to construct the streamspace representation of higher-level data structures. These types are categorized into element-manipulating types (Table 2.1) and stream-creating types (Table 2.2).

The element-manipulating types only manipulate the size of the element (i.e. they can be mapped to a one-dimensional bitvector). $Bits\langle B\rangle$ is a leaf node that adds B bits to the element, indicating a transfer of B bits. $Group\langle S_1, S_2, ..., S_n\rangle$ concatenates the child elements into a single field, resulting in an element size that is the sum of all the child element sizes, if all the elements are in the same physical stream (i.e. there are no nested stream nodes inside $Group\langle S\rangle$). $Union\langle S_1, S_2, ..., S_n\rangle$ creates an element with its element size set to the widest element size across the child types. The $Group\langle\rangle$ and $Union\langle\rangle$

nodes are not limited to element-manipulating types, therefore they can be used to combine multiple physical streams.

The stream-manipulating types are used to define new physical streams. $New\langle S\rangle$ can be used as a root node, or to create a new physical stream that has the same dimensionality as its parent. $Dim\langle S\rangle$ increases the dimensionality of its child type. And finally, $Rev\langle S\rangle$ creates a physical stream that flows in the reverse direction with respect to its parent.

Type	**Description**
$Bits\langle B\rangle$	Primitive element with B bitwidth.
$Group\langle S_1, S_2, ..., S_n\rangle$	Concatenates elements of types $S_1, S_2, ..., S_n$ into one physical stream element.
$Union\langle S_1, S_2, ..., S_n\rangle$	Creates a B-bits element, where B is the max. element with of $S_1, S_2, ..., S_N$.

Table 2.1: Tydi basic element-manipulating nodes.

Type	**Alias**	**Description**
$Stream\langle T_e, t, 1, Sync, c, Forward, T_u, false\rangle$	$Dim\langle T_e, t, c, T_u\rangle$	Creates a streamspace of T_e in the next dimension w.r.t. its parent.
$Stream\langle T_e, t, 0, Sync, c, Forward, T_u, false\rangle$	$New\langle T_e, t, c, T_u\rangle$	Creates a new physical stream in the current dimension.
$Stream\langle T_e, t, 0, Sync, c, Reverse, T_u, false\rangle$	$Rev\langle T_e, t, c, T_u\rangle$	Creates a new physical stream of T_e in reverse direction w.r.t. its parent.

Table 2.2: Tydi basic stream-manipulating nodes.

2.3.2. Container library

Tydi proposes a standard container library to define the representation of common data structures in streamspace. The relevant containers can be found in Table 2.3. These containers server a similar purpose as the containers defined in many programming language's standard libraries: aliases for combinations of types from the type system that represent a certain access pattern.

Data type	**Tydi container**	**Definition**
$Empty$	$Null$	$Bits\langle 0\rangle$
$Prim\langle B\rangle$	$Bits\langle B\rangle$	$Bits\langle B\rangle$
$Struct\langle T_1, T_2, ..., T_n\rangle$	$ConcatStruct\langle S_1, S_2, ..., S_n\rangle$ $DesyncStruct\langle S_1, S_2, ..., S_n\rangle$	$Group\langle S_1, S_2, ..., S_n\rangle$ $Group\langle New\langle S_1\rangle, New\langle S_2\rangle, ..., New\langle S_n\rangle\rangle$
$Variant\langle T_1, T_2, ..., T_n\rangle$	$PackedVariant\langle S_1, S_2, ..., S_n\rangle$ $ConcatVariant\langle S_1, S_2, ..., S_n\rangle$ $DesyncVariant\langle S_1, S_2, ..., S_n\rangle$	$Group\langle Bits\langle\lceil log_2 2\rceil\rangle, Union\langle S_1, S_2, ..., S_n\rangle\rangle$ $Group\langle Bits\langle\lceil log_2 2\rceil\rangle, Group\langle S_1, S_2, ..., S_n\rangle\rangle$ $Group\langle Bits\langle\lceil log_2 2\rceil\rangle, New\langle S_1\rangle, ..., New\langle S_n\rangle\rangle$
$Seq\langle T\rangle$	$List\langle S\rangle$ $Vector\langle S\rangle$	$Dim\langle S\rangle$ $Group\langle Bits\langle L\rangle, New\langle S\rangle\rangle$

Table 2.3: Tydi container types.

2.3.3. Physical streams

The streamspace types can be mapped into physical streams with the following parameters:

$$PhysicalStream(E, N, D, C, U)$$

where:

E – element bits
N – elements per transfer
D – dimensionality
C – complexity level
U – user bits

The element bits (*E*) and dimensionality (*D*) is determined by the streamspace type. The additional parameters are relevant in controlling the throughput of the stream and determining compatibility between two interfaces.

N determines the number of elements to be transferred in one transaction. By communicating multiple elements per cycle, the throughput of the stream can be scaled. N can be derived from the *throughput ratio* of the logical stream, as it equals $\lceil \prod t \rceil$ for all the ancestral nodes.

C is the complexity level of the particular stream. The complexity level determines the guarantees made by the source and describes how the elements are transferred in streamspace. It can be used to make trade-offs regarding the control logic on both ends of the stream.

Signal name	Bitwidth	Function
$valid$	$scalar$	Handshaking
$ready$	$scalar$	
$data$	$N \times \lvert E \rvert$	Data lanes
$last$	$N \times D$	Indicating the last transfer in the signaled dimensions
$stai$	$\lceil log_2 N \rceil$	Index of the first valid lane
$endi$	$\lceil log_2 N \rceil$	Index of the last valid lane
$strb$	N	Individual lane validity mask
$user$	$\lvert U \rvert$	User-defined additional control information

Table 2.4: Tydi physical stream signals

The signal composition of a physical stream can be seen in Table 2.4. The presence and bitwidth of some signals are dependent on the complexity level of the stream. The detailed discussion can be found in the online specification [8].

2.3.4. Streamlets

In a dataflow design, transformations are typically implemented as *streamlets*, which are components with streaming interfaces. An example streamlet with Tydi interfaces is shown in Figure 2.4. The streamlet takes a list of strings on the input which is represented as $Stream\langle Bits\langle 8\rangle, d = 2, t = 20\rangle$ in streamspace. The $Bits\langle 8\rangle$ element-manipulating node represents an ASCII character, the $d = 2$ parameter defines the stream to be two-dimensional (a list of strings is modeled as a two-dimensional sequence of characters), and the $t = 20$ parameter ensures that 20 characters are transferable in every cycle. The output is a one-dimensional, 64-bit integer stream. From this specification, the HDL template of the streamlet with the physical streams can be generated in the target language, in this case VHDL.

```
Streamlet ExampleStreamlet (
    strings: in Stream<Bits<8>, t=20, d=2>,
    char_count: out Stream<Bits<64>, d=1>)
```

(a)

```
component ExampleStreamlet_com
  port(
    clk              : in std_logic;
    rst              : in std_logic;
    strings_valid    : in std_logic;
    strings_ready    : out std_logic;
    strings_data     : in std_logic_vector(159 downto 0);
    strings_last     : in std_logic_vector(1 downto 0);
    strings_endi     : in std_logic_vector(4 downto 0);
    strings_strb     : in std_logic_vector(19 downto 0);
    char_count_valid : out std_logic;
    char_count_ready : in std_logic;
    char_count_data  : out std_logic_vector(63 downto 0);
    char_count_last  : out std_logic_vector(0 downto 0);
    char_count_strb  : out std_logic_vector(0 downto 0)
  );
end component;
```

(b)

Figure 2.4: (a) Streamlet definition. (b) Streamlet as a VHDL component.

3 Solution architecture

Despite a seven order of magnitude increase in transistor count, backed by Moore's law since the '70s, the hardware design methodology mostly stayed the same. Digital hardware is predominantly developed in RTL, using languages (namely, VHDL and Verilog) that couple behavior, timing, and target-specific constraints. With the ever-increasing complexity, RTL descriptions lead to verbose and error-prone designs. Consecutively, the complexity of verification also increased, making it problematic to reach sufficient test coverage.

Several HLS (high-level synthesis) tools appeared to raise the level of abstraction by approaching hardware design from a software background. However, software constructs generally lack crucial architectural information about the design. Therefore, those decisions have to be made later by the tool or guidance is required from the developer. Due to the substantial semantic gap between the software-like description and hardware, the resulting design's performance is heavily dependent on the capabilities of the tool and the model's fit for hardware, which is ultimately determined by the application profile and the developer. To be able to exploit the capabilities of the target hardware platform and tools, extensive knowledge is usually required about the hardware to formulate the description in a way that the HLS tool of choice is capable of synthesizing a design that meets the target metrics.

Recently, new languages and frameworks emerged that are approaching hardware design from unconventional abstraction levels and trying to find middle-ground between low-level HDLs and HLS.

This chapter gives insight into some of the promising hardware description/composition languages and synthesis tools with special attention to their advantages and limitations for accelerator kernel development in a big data context, and their integrability into a workflow with Arrow and Fletcher (Tydi).

3.1. Modern hardware description languages

Experts argue that we're not only in the "golden age of computer architectures", we're also in the "golden age of hardware description languages" [39], driven by the productivity increase desired by hardware architects. They observe a trend that more advanced designs can be synthesized from fewer lines of code. Furthermore, the hardware community is starting to apply software programming language techniques, such as meta-programming, polymorphism, and abstract data types to address some of the productivity bottlenecks.

The general trend in modern HDLs is to raise the abstraction level, allow greater generalization and increase IP reusability. In this section, we look at some of the new and promising developments in the field.

3.1.1. eDSLs for hardware description

There is an emerging subclass of languages that embed their circuit abstractions into a host-language, leveraging its meta-programming features. These languages are often called Hardware Construction Languages (HCLs). Chisel, SpinalHDL, MyHDL, etc. all fall into this category, with differences in feature set and host-language. Arguably, the most common structural language from this class is Chisel [13], with Scala as its host-language.

The advantage of using an eDSL (embedded domain-specific language) is that the hardware primitives can be manipulated using the constructs of the host-language. In the case of Chisel, the Scala-embedding raises the abstraction level of circuit design by providing concepts including object-orientation, functional programming, parameterized types, and type inference.

In order to build performant accelerators, a number of microarchitectural decisions have to be made. These parameters are generally determined during a design-space exploration process, but traditional HDLs lack the necessary features to describe hardware with adequate generality to compose hardware during these processes. A common approach to overcome these limitations is to use code generators or macro preprocessors. By relying on an eDSL-based development flow, the necessary abstractions can be built to capture high-level design patterns required for productive accelerator development.

Diplomacy [15] and RocketChip [13] are prominent examples for Chisel. RocketChip is a SoC generator, while Diplomacy is a parameter negotiation library in Scala. Diplomacy is used by RocketChip to negotiate parameters while connecting components to a shared interconnection network. This level of automation is made possible by the strong support for parametric designs in Chisel, and by the tight integration with a powerful host-language.

Simulation: VHDL and Verilog were originally developed as *hardware simulation* languages, and only later, a subset of the language features were adopted for *synthesis*. However, with increasing design complexity, simulation also became cumbersome, especially in the acceleration field where verification involves complex testbenches to model the host-side application or interact with it in a transparent way.

Therefore, the need emerged for simulation solutions that allow developers to build testbenches in a higher-level language, possibly in a more software-oriented way. There are promising open-source solutions to tackle these challenges, like Cocotb [6] and Verilator [9]. Cocotb is a Python library for coroutine based co-simulation of VHDL and Verilog designs. It allows the developer to write testbenches in Python, making it possible to use Python's language features to provide stimulus to the design or to interface with other languages. Verilator, on the other hand, compiles synthesizable Verilog designs into cycle-accurate C++ code and provides an API to interact with the design. Verilator is one of the fastest Verilog simulators on the market.

The former solutions focus on raising the abstraction of simulation testbenches and enabling integration for Verilog and VHDL designs.

Chisel also has advantages on the simulation front. By being an eDSL in Scala, the full language feature set is available for verification as well. By wrapping Chisel's low-level simulation API (*PeekPokeTester*), all the necessary abstractions can be built to develop and execute the simulation on the desired level. Also, by developing the testbench in Scala, the verification can be seamlessly inserted into continuous integration pipelines without using a number of external tools.

Chisel's simulation offerings are also widened by Firesim [24], which is a cycle-accurate, FPGA-accelerated simulation platform that runs on Amazon AWS F1 instances.

Dataflow design: Although Chisel is not specifically built for dataflow-oriented hardware design, it has basic support for streaming dataflow design by abstracting away the handshaking for custom types using the DecoupledIO class. With the possibility of building more abstractions, dataflow designs can be built in a parametric and productive way, without writing verbose code.

Intel has already published a framework [12] to support dataflow design for accelerators by providing basic components:

- Load/store units
- Memory arbiter
- Type packer/unpacker
- SDF (static dataflow) actor

All the components use DecoupledIO to abstract away the handshaking between the components. Using these units, accelerators can be built in a convenient way by wrapping the user logic into SDF actors.

Chisel supports user objects through the Bundle class, which is similar to a VHDL record. These types can be used as custom data types between components:

```
new AccUserIn(new UserBundle, 16, BUF_SIZE)
```

In Intel's framework, the functionality of the accelerator is described in the context of SDF actors. It is an abstraction that implements a dataflow actor that supports multiple inputs and outputs (DecoupledIO interfaces). It fires when all the inputs are valid, and all the outputs are ready. These SDF actors can be composed into a dataflow design and integrated into the acceleration flow using the memory components.

Applicability: Chisel has major advantages compared to traditional HDLs that mostly come from the parametrization and abstraction capabilities. These features make IP reuse significantly more convenient, which is a must for accelerator development, where a high level of flexibility is required to handle a variety of use-cases.

The increased productivity is a major factor that contributed to the success of eDSLs, but in the big data field having the advantage of building the whole design in a powerful host-language is invaluable. We can take the initial Apache Arrow + Fletcher workflow as an example. As described in Chapter 2, the developer presents an Arrow RecordBatch to Fletchgen (C++ tool) that generates VHDL with the necessary components, a simulation testbench and a skeleton for the user kernel. Next, the developer designs the kernel and simulates it with one of the supported tools (GHDL, ModelSim, etc.). The simulation testbench is based on the RecordBatch that was provided during the Fletchgen run.

This workflow could be developed in Chisel, under one umbrella. The Apache Arrow-related processing using the Java Arrow API, the hardware generation, and simulation using Chisel.

If we limit the scope for kernel development, Chisel would be ideal for hand-built, performance-oriented, parametric designs, or in use-cases where otherwise a code-generator would have to be used (e.g. a regular expression matcher generator).

3.1.2. Transaction-Level Verilog

TL-Verilog (Transaction-Level Verilog) is a unique concept across the new hardware modeling languages. It expresses behavior as timing-abstract pipelines [18], allowing the developer to build control-intensive applications with cycle-level detail, but keeping the exact staging of the design subject to change.

Timing-abstract pipelines: In TL-Verilog the behavior is specified withing the context of pipelines in a timing-abstract manner. An example design is shown in Figure 3.1 and Figure 3.2.

TL-Verilog abstracts away timing by introducing *pipesignals* (such as $\$aa$) that live in the context of *pipelines* ($|calc$) and *pipestages* ($@1$). Pipesignals represent the signal and its staged versions. Where pipesignals cross pipestage boundaries, the necessary sequential elements are inserted by the tool. The staging is considered to be a physical attribute; thus, retiming does not change the behavioral model, it just describes one possible implementation in the design space.

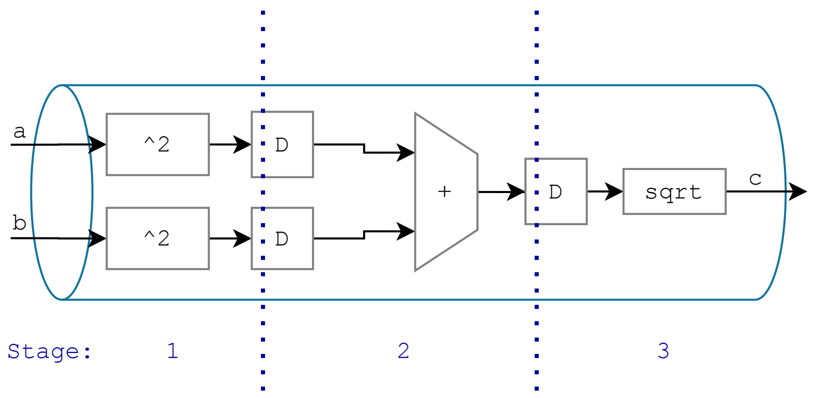


Figure 3.1: Hardware block diagram of the Pythagorean theorem calculator pipeline. [18]

```
|calc
  ?$valid
     @1
        $aa_sq[7:0] = $aa[3:0] ** 2;
        $bb_sq[7:0] = $bb[3:0] ** 2;
     @2
        $cc_sq[8:0] = $aa_sq + $bb_sq;
     @3
        $cc[4:0] = sqrt($cc_sq);
```

Figure 3.2: Pythagorean theorem calculator pipeline example in TL-Verilog. [18]

Transaction flow: TL-Verilog also supports building designs from flow components by extending timing-abstraction to transactions flowing through arbitrary components, including FIFOs, arbiters, and queues. [19]

An example transaction flow design is shown in Figure 3.4. The design instantiates and connects 4 identical logic blocks in a ring architecture. Each cycle, each of the logic block are allowed to generate a transaction that will travel the ring to its destination.

In TL-Verilog, a transaction is a collection of signals, or *fields*. The transaction-level design is supported in TL-Verilog by the $ANY construct, which can be used to abstract away the signals that are passed through the component. (Figure 3.3)

```
$ANY = $select ? /in1 $ANY : /in2 $ANY;
```

Figure 3.3: Example flow expression for a multiplexer. [19]

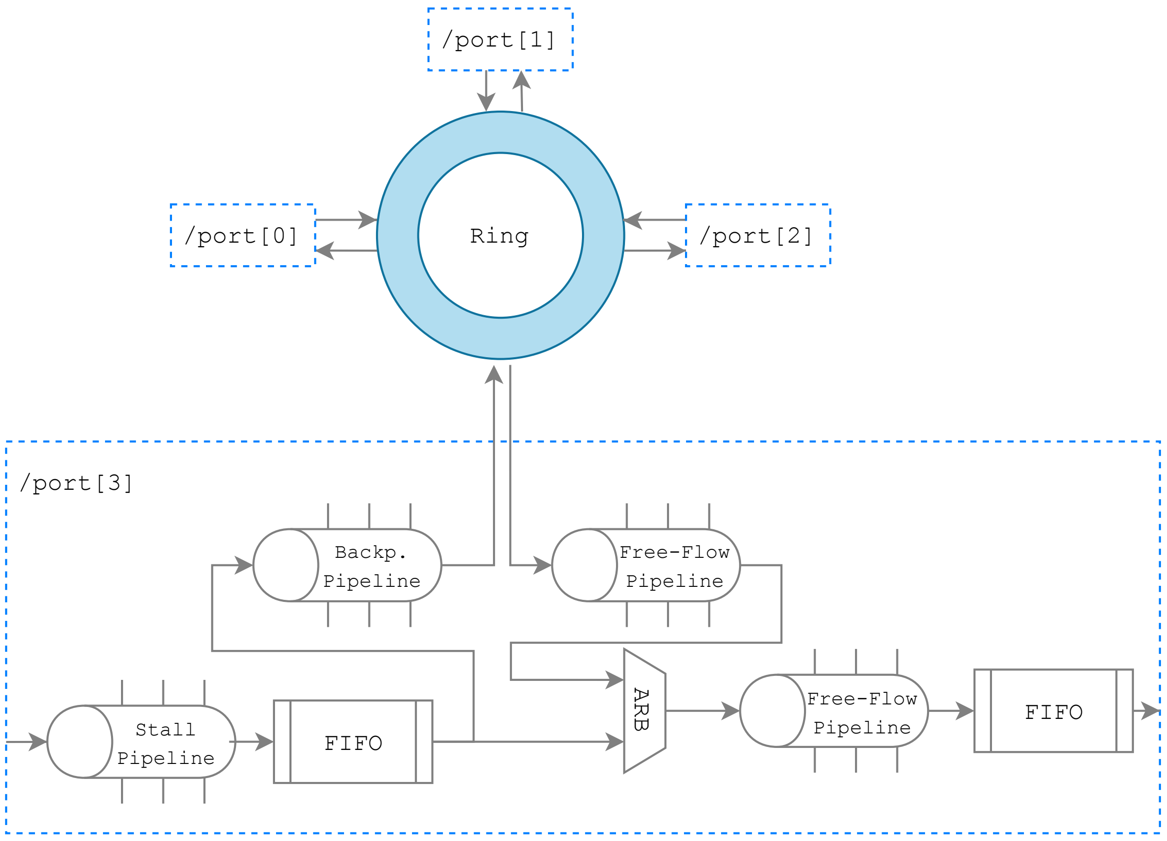


Figure 3.4: TL-Verilog transaction flow example. [19]

Applicability: In the context of acceleration, TL-Verilog is beneficial for building complex, control-heavy processing modules. It is also suitable for building microarchitectures using general flow components. The retiming capabilities are attractive in case certain clock speed, performance and area targets have to be met and are subject to change.

3.2. Synthesis frameworks

This section introduces frameworks that operate on unconventional levels of abstraction, making them attractive to the target application domain.

Parallel patterns: Parallel patterns are constructs in functional languages that are implicitly parallel, such as *map*, *filter*, *reduce*, and *groupBy*. Applications that can be efficiently expressed using parallel patterns due to their spatial and temporal locality are the ones that can benefit most from FPGA acceleration. These applications are common in image processing, financial analytics, machine learning, and big data analytics. [34]

3.2.1. Spatial

Spatial [26] (successor of DHDL [25]) is a domain-specific language hosted on Scala that specifically targets application accelerators. Although Spatial is considered to be an HLS tool, it features hardware-centric abstractions.

Spatial as a language: Spatial is a timing abstract language with abstract control sequences: *FSM*, *Foreach*, *Reduce*, *Memreduce*, *Stream*, and *Parallel*. These control structures can be arbitrarily nested to build hierarchical pipelines and exploit nested parallelism. Spatial also borrows constructs from parallel patterns by providing a compact syntax for reductions that implies associativity, allowing the compiler to parallelize. The *Foreach*, *Reduce* and *MemReduce* constructs can be parallelized by applying parallelization factors. Spatial guarantees that the parallelization maintains the same behavior as the sequential execution. The bodies of the control structures are untimed, the operations are scheduled by the compiler.

In Spatial, explicit control is given to the developer over the memory hierarchy by providing a variety of memory templates, including on-chip scratchpads (*SRAM*), line buffers (*LineBuffer*), queues and stacks (*FIFO* and *LIFO*), registers (*Reg*), register files (*RegFile*) and read-only lookup-tables (*LUT*). The *DRAM* template represents the highest level of the memory hierarchy, it can be accessed using pre-defined access patterns: *load-store* and *scatter-gather*. Spatial offers a variety of interfaces to the host, including register IO (*ArgIn*, *ArgOut*, *HostIO*), external streaming interfaces for peripherals (*StreamIn*, *StreamOut*), and a high-bandwidth datapath through the DRAM.

The framework allows the developer to write the host application and the accelerator code in the same Scala project. The partitioning of the host-application and the accelerator code is done using the *Accel* scope. All operation within this scope will be allocated to the accelerator, while the rest will run on the host. As in data types, Spatial supports statically-sized aggregate types in hardware, but lacks inherit support for dynamically sized, or multidimensional aggregate types.

An example Spatial application is shown in Figure 3.5. It demonstrates the basic structure of a Spatial application through the calculation of a dot product.

Spatial as a compiler: Since Spatial is a timing-abstract language, scheduling is done by the compiler. Where the controller type is not specified explicitly, the compiler will infer one. After the controller insertion, the compiler will attempt to schedule the operations within the controllers.

Spatial supports loop unrolling and pipelining. These optimizations require support in the memory hierarchy; therefore, Spatial is capable of partitioning, banking, and buffering on-chip memories to be able to serve the necessary bandwidth.

After the scheduling and memory banking options are determined, together with loop parallelization and tile size parameters, the compiler performs an area and runtime estimation to be used during the design space exploration, for which, Spatial employs a self-learning autotuner called HyperMapper. The parameters can be implicit pipelining and parallelization parameters, or explicitly included in the code.

```
val output = ArgOut[Float]
val v_a = DRAM[Float](N)
val v_b = DRAM[Float](N)
Accel {
  Reduce(output)(N by B){ i =>
    val tile_a = SRAM[Float](B)
    val tile_b = SRAM[Float](B)
    val res = Reg[Float]

    tile_a load v_a(i :: i+B)
    tile_b load v_b(i :: i+B)

    Reduce(res)(B by 1){ j =>
      tile_a(j) * tile_b(j)
    }{a, b => a + b}
  }{a, b => a + b}
}
```

Figure 3.5: Spatial dot product example

After the selection of design parameters, the design gets finalized by performing the unrolling and retiming based on the prior analysis passes.

Finally, the output product is generated, which is Chisel RTL. The code generator instantiates modules from a library of parameterized templates written in Chisel, and generates to glue logic to tie them together.

Spatial as a framework: Spatial is an end-to-end framework, meaning that the Spatial compiler synthesizes the accelerator, and also manages the integration with the host-side software. The high-level hardware architecture of the framework is depicted in Figure 3.6. The centerpiece of the final hardware design is the accelerator core generated by the framework in Chisel. The accelerator core is wrapped in a *Fringe*, which is also a Chisel design with the purpose of interfacing the generated core with the target hardware platform by connecting the register, streaming and memory interfaces and instantiating platform-agnostic modules.

The framework currently supports the following target platforms: Xilinx Ultrascale+ VU9P FPGAs on Amazon's EC2 F1 Instances, Xilinx Zynq-7000 and Ultrascale+ ZCU102 SoCs, Altera DE1, and Arria 10 SoCs.

Applicability: Since Spatial is an end-to-end framework, it is not built to be used as a tool to generate stand-alone kernels. In particular, the desired Apache Arrow + Fletcher workflow is redundant with Spatial's host-side runtime and accelerator interface.

However, using the *StreamIn/Out* interfaces, it would be possible to include a Spatial design into a custom dataflow. For this the following components would be required:

- Fringe (Chisel)
- Generic Chisel stream components
- Spatial internals to support custom components

With these modifications, Spatial would be able to interface with non-dimensional Tydi streams that encapsulate statically-sized aggregate types. By using Spatial this way, major advantages would be lost, including the rich memory optimization features and DSE. However, a similar DSE approach could be used to determine an appropriate elements-per-cycle value for the incoming streams.

Not considering the redundant feature set, Spatial represents a unique abstraction level that is ideal for computational kernels and appeals to developers both with software or hardware backgrounds.

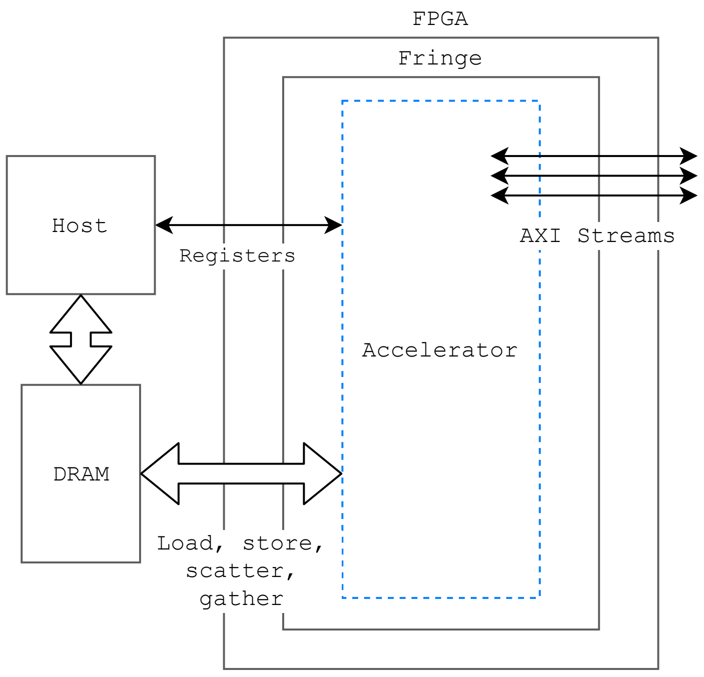


Figure 3.6: Spatial framework top-level hardware architecture overview.

3.2.2. Lift

Lift [27, 37] is a framework that claims to achieve performance portability across heterogeneous systems. The applications are described using parallel patterns, in a platform-agnostic language. The compiler then generates platform-specific code after a design space exploration driven by a set of rewrite-rules. The rewrite-rules are used to express platform-specific optimization choices, including algorithmic and low-level hardware optimizations.

FPGA backend: Lift introduces a set of low-level patterns that can be synthesized to FPGAs. The framework represents the design internally as dataflows; hence, it is possible to synthesize the design as streaming components connected in a consumer-producer fashion. Lift's unique feature is that it supports multidimensional streams on FPGA by having a multi-bit *last* signal. As in data types, fixed-size scalar types are supported.

The low-level pattern used by Lift to support FPGA synthesis:

- **ZipStream:** ZipStreams takes two streams and creates a stream of tuples.
- **LetStream:** LetStream takes a function and instantiates it next to a Block RAM cache. It allows the function to consume the input data multiple times.
- **MapStream:** MapStream takes a function declaration, and applies that function to every element of the stream.
- **ReduceStream:** The ReduceStream pattern reduces a sequence of multiple elements into a sequence of one element.
- **SplitStream:** SplitStream introduces a new inner dimension to a stream given the size of the inner dimension.
- **JoinStream:** JoinStream removes the outermost dimension from a stream by discarding the corresponding *last* signal.
- **UserModule:** User-defined module that operates on streams without requiring any global context.

```
add = fun(x => UserModule.Addition(x))
mul = fun(x => UserModule.Multiplication(x))
program = fun( (x, y) =>
  ToHost(
    LetStream(fun (z =>
      ReduceStream(add , 0,
        MapStream(mul , ZipStream(z, ToFPGA(y)))
      ))
      , ToFPGA(x)
)))
```

Figure 3.7: Matrix multiplication in Lift low-level patterns. [27]

An example application is shown in fig. 3.7. The program performs a dot product calculation and it is described as nested function calls. It starts with the *ToFPGA* pattern that sends argument *x* to the FPGA. This argument is stored using the *LetStream* pattern. Then, the second argument is sent and zipped with the first one. These arguments are pairwise multiplied and reduced using an addition operation. Finally, the result is returned using the *ToHost* construct. The addition a multiplication operators are represented using the *UserModule* construct which can be implemented as a separate entity. The nested function calls of the Lift lift program can be translated into a dataflow design and composed using hardware streams and the described Lift low-level patterns.

Applicability: The Lift framework proposes a unique approach for targeting multiple hardware architectures from the same application expressed in parallel patterns. The authors evaluated the proof-of-concept approach of targeting FPGAs on a Xilinx XC7Z010 device and their approach showed a 10x improvement in matrix multiplication compared to the CPU implementation and a commercial HLS tool.

Since the Lift language relies heavily on parallel patterns, it is a good fit for describing common algorithms that appear in the big data field. However, the framework has not been evaluated on a large-scale acceleration platform. In addition, the data type support is currently limited.

The proposed patterns are a good foundation to build on, and to extend them for more complex data types. These patterns were taken into consideration during the process of defining parallel pattern-like constructs for Tydi.

3.2.3. Fleet

Fleet [38] is a recently published framework for parallel streaming processing on FPGAs. Fleet expects the developer to provide a processing unit that takes a single stream of tokens, which is then instantiated in as many copies as the user wants, along with a soft memory controller that feeds and drains the units. (Figure 3.8)

Processing units: The processing units have to be provided in RTL, with ready-valid interfaces on the input and the output. The units can be written by hand or generated by higher-level tools and should serially process a single stream of tokens.

Fleet eDSL: Fleet also provides a Scala eDSL as an extension to Chisel to write processing units. The Fleet language abstracts away the handshaking and provides the user with automatically pipelined BRAM type.

The basic language features include registers, binary operators, and conditional blocks. In addition, Fleet defines the *input* keyword to provide access the current input token, *emit* to produce an output token, and a *while loop* construct to take multiple virtual cycles for the current input token. The statements in the language can be contained in *if*, *else if*, and *else* conditional blocks.

As in data types, Fleet eDSL supports tokens with fixed width defined in compile-time. Operators are provided for integer and boolean types.

To allow automatic BRAM pipelining, Fleet eDSL introduces the *virtual cycle* abstraction. Every execution of the user's processing logic is one virtual cycle, and Fleet guarantees that a virtual cycle is one real hardware cycle. To ensure this, Fleet restricts the BRAM access pattern in a single virtual

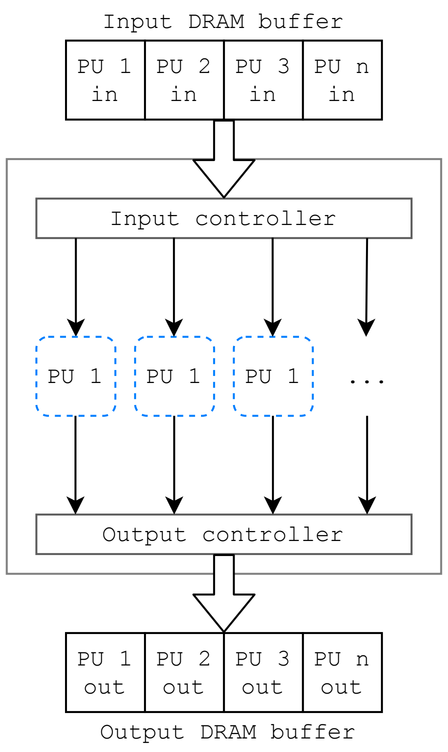


Figure 3.8: Fleet framework hardware overview. [38]

cycle in the following ways: dependent reads are not allowed, the algorithm is only allowed to read the BRAM at one address and write it at one address, and finally, only one token is allowed to be emitted.

If these restrictions are met, the Fleet compiler can generate a two-stage pipeline for the virtual cycles, with one stage for writes, and one stage for reads.

In summary, the Fleet eDSL allows users to design stream processing units in a productive way, while keeping full control over the performance of the resulting hardware. An example stream filter processing unit is shown in Figure 3.9.

```
class Filter(coreId: Int) extends ProcessingUnit(8, 8, coreId) {
  onInput {
    swhen(StreamInput > 127.L) {
      Emit(StreamInput)
    }
  }
  Builder.curBuilder.compile()
}
```

Figure 3.9: Simple stream filter example in Fleet eDSL. [7]

Memory controller: Fleet provides a soft memory controller with the design that feeds and drains the processing units in a round-robin fashion.

The Fleet memory controller utilizes an AXI4 memory interface with a 512-bit data bus and an arbitrary number of AXI4-channels. The controller operates by issuing DRAM requests at the granularity of multiples of the data bus width. The processing units have BRAM input and output buffers with enough capacity to store a burst. Since the input and output buffers of the processing units are usually smaller than the AXI4 data bus width, burst registers are used to feed and drain data from multiple processing units at once.

On the host-side, the user is expected to prepare a contiguous buffer with the input data, which will be streamed into the accelerator DRAM and distributed among the processing units. The result are copied back in a similar way to the host.

Applicability: Fleet offers a massively parallel streaming abstraction to scale accelerator performance, and a Scala eDSL for building arbitrary streaming pipelines, both of which are valuable for building high-throughput accelerators. The proposed eDSL is very close to an envisioned abstraction level for building dataflow kernels in a hardware-oriented way. Although the framework supports custom, statically sized data types for tokens, it lacks support for nested dynamically-sized data structures.

Fleet is a proof-of-concept with great potential, and could be used to accelerate a number of processing tasks in a big data environment. However, to become a more general solution, it needs to incorporate support for more complex data types, and improve host-side integration, possibly by leveraging standard in-memory formats, like Apache Arrow.

3.3. End-to-end frameworks for big data

This section introduces two frameworks that are specifically built for big data applications. These solutions are on the other extreme of the spectrum compared to hand-built accelerators, and show the level of complexity associated with bridging the huge semantic gap between the application and the target execution platform.

3.3.1. Melia

Melia [40] is an OpenCL-based framework to build MapReduce [16] applications for FPGAs. It extends on the idea of FPMR [36], but in Melia, the custom data processing tasks are implemented in OpenCL instead of a low-level hardware description language like VHDL/Verilog.

Melia is implemented as a software library, and can be used in a similar way as other MapReduce frameworks, specifically, users are required to implement the *map()* and *reduce()* transformation functions in C. Melia then synthesizes these functions and executes them.

The framework proposes optimizations that are attractive for FPGAs, such as memory coalescing, pipeline replications, and loop unrolling. The design space exploration is done using a cost model proposed by the authors.

The comparisons between FPGA and CPU/GPU show a steady improvement in energy efficiency, being 3.6 times more efficient than the GPU implementations, and 16.7 times more efficient than the CPU implementations in the benchmarked applications (K-means clustering, Word count, Distinct words, String matching, Matrix multiplication, and Similarity scope).

3.3.2. S2FA

S2FA (Spark-to-FPGA-Accelerator) [42] is a framework that generates FPGA designs directly from Apache Spark applications, using Blaze [22] as a runtime backend. The framework is able to compile user-written Scala code as long as the function satisfies certain constraints related to data types, memory allocation, and library calls.

S2FA follows a practical approach to generate the FPGA kernels by generating C code from Java bytecode which is then advanced using the Merlin transformation library and fed into an HLS tool (SDAccel) to generate an FPGA design.

The cornerstone of the S2FA framework is the design space exploration, since it has to close a remarkable semantic gap between the user-written Scala code and generated hardware. The framework employs a learning-based exploration that is built on top of OpenTuner [11]. It works with multiple reinforcement learning algorithms simultaneously and adopts a multi-armed bandit algorithm to judge the effectiveness of each technique.

3.4. Discussion

In this section, we looked at various solutions for FPGA accelerator kernel development and synthesis with a big data focus. The introduced solutions range from low-level HDLs to end-to-end frameworks that are already integrated with data analytics packages or represent one of the popular programming models in the application domain: parallel patterns.

Every evaluated solution has its place in accelerator kernel development: HDLs for high-performance, tailored implementations, synthesis frameworks to raise the abstraction level for increased productivity, and end-to-end frameworks for plug-in solutions.

However, none of the described solutions have the necessary integration-level and flexibility to develop performant applications with low turnaround time and adequate control over the hardware. Without leveraging standardization on the host-side and on the accelerator-side, the re-use of accelerator cores is challenging, and results in an ad-hoc mix of software and hardware components.

By looking at the whole spectrum of solutions, we can observe a lack of cross-compatibility. The big data field utilizes a broad range of algorithms, such as (de-)compression, image processing, machine learning, graph analytics, sequence alignment, statistical analytics, and text processing. In order to cover this broad range of distinct areas, it would be beneficial to have the ability to combine components from different sources, or ones that are written in different languages/frameworks.

In data analytics, complex data types are common. As we observed, handling these types is challenging in most languages; furthermore, parts of the data structure or the dimensionality information may not be related to the processing task. As a general design objective, it is desirable to write components in a way that they contain functionality and control logic only related to the processing task, similar to a function in software development. However, to build complex designs using components designed this way, a solution is required to handle some of the complexity associated with compound and multidimensional data types outside the processing units.

Currently, there is no language or framework that satisfies all these constraints. Fletcher proposes to solve the standardized software-side integration, Tydi proposes to provide a standard interface between components, but at present, there is no solution to describe hardware compositions from components that implement these interfaces.

3.5. Proposed solution

The proposed solution in this thesis is a composition tool for components that implement Tydi interfaces, with higher-level language features analogous to the low-level patterns introduced by the Lift framework.

A practical example: Assume a column in a dataset that holds the last names of people. The processing task is to capitalize all of these names. In this case, the data is a list of strings, that can be modeled in Tydi streamspace in the following way: $Seq\langle Seq\langle Prim\langle 8\rangle\rangle\rangle$. The atomic operation in this processing task is the capitalization of individual characters. Assuming no help from higher-level patterns, the streamlet that performs the capitalization would have to implement the above input and output type, even though the dimensionality information is not relevant to the operation. To tackle this issue, one would want to peel off the outer dimensions, perform the operation on the individual characters, and construct an output stream with the new elements and with the same dimensionality as the input was. The described operation resembles the MapStream pattern introduced in Lift.

Using a MapStream pattern defined for Tydi, the description would look like the following:

```
map_strings: MapStream(map_characters: MapStream(op_inst: ops.Capitalize));
```

A streamlet described as above would instantiate the *Capitalize* streamlet from the *ops* library inside a Tydi project, the two nested MapStream primitives would advance the stream to make it two-dimensional, so the kernel would have the desired input and output type.

The proposed approach allows the developer to combine components that are written in different languages into more complex designs, and increase productivity using the higher-level primitives. Assuming a set of common operations and accelerator kernels for specific workloads, application-specific accelerators could be written effectively and effortlessly, while keeping the potential for low-level optimizations. A composition language could also become a solid basis for generating kernels by higher-level tools, incorporating pre-built and synthesized kernels without the need to produce the complete hardware description.

4
Prototyping

4.1. Composition language

This section discusses the proposed composition language for Tydi. The grammar specification is available as a PEST [3] PEG file in Appendix A. PEST is a parser framework written in Rust, it uses parsing expression grammars (PEGs) as input, making it a productive solution for prototyping.

4.1.1. Implementation of a streamlet

Tydi currently has means to define a streamlet with its interfaces and insert it into a project hierarchy but has no way to attach an implementation. This section proposes language constructs specifying the implementation of a streamlet in the project hierarchy.

HDL: The implementation of a streamlet can be specified as one of the supported HDL languages (currently Chisel and VHDL). In this case, streamlet skeletons should be generated in the target language with the appropriate Tydi interfaces.

```
impl lib.Streamlet VHDL
```

External: The implementation of a streamlet can be specified as ”external”. In this case, the implementation of the streamlet already exists externally, or has to generated by an external tool. A command string can be specified that will be executed during the implementation phase. The command can include parameters that can be assigned during instantiation.

```
impl lib.Streamlet external ”command string with $parameters”
```

Structural: As the main feature of the proposed language, streamlets can be constructed in a structural way by instantiating streamlets from the project or pattern nodes and connecting them.

```
impl lib.Streamlet structural {
 /*instantiations and connections*/
}
```

Instantiation: The instantiation of a streamlet is defined as follows:

```
instance_name: library.Streamlet[param1 := 6, param2 := ”string parameter”,
  param3 := false];
```

The instantiation consists of an instance name, a reference to the streamlet to be instantiated, and a list of parameters. The supported parameter types are: *string*, *number*, and *boolean*.

The parameters should be passed to the implementation backend of the streamlet. In the case of an HDL backend, the parameters should be declared and assigned as regular generics, while in the case of an external implementation, these parameters should be used in the command string.

Single connection: A single connection between two interfaces is defined as follows:

```
instance1.interface_a <= instance2.interface_b;
```

Where the left side is always an interface defined as an input and the right is defined as an output. The *this* keyword can be used to access the interfaces of the streamlet to be implemented from the inside.

Using the *this* keyword, the streamlet's interfaces appear in the reverse direction, so the input and output connections can be made in the following way:

```
instance.in <= this.in;
this.out <= instance.out;
```

Before a connection is registered, the compatibility between the source and destination interface has to be validated based on the type and complexity level. If the type and complexity level allows it, conversion logic may be inserted.

Chain connection: Streamlet instances can be chained together using the chain connection construct in the following way:

```
inst1 <=> inst2 <=> inst3;
```

The construct connects the primary input and output interfaces of two streamlets in a more visual way, without the need for two single connections. Since currently, there is no way to specify primary interfaces if there is more than one input or output, the interfaces with the names *in* and *out* are connected.

Unconnected interfaces: The unconnected output interfaces should be connected to a dummy sink that has its *ready* signal asserted at all times to prevent stalling.

4.1.2. Parallel patterns

This subsection defines higher-level primitives that can be applied to Tydi streams and allow the construction of designs that operate on complex stream types from simple streamlets. These pattern nodes have to be generated during the build process.

MapStream

The MapStream pattern removes the outermost dimension of the input stream, feeds it into a streamlet, and adds back the removed dimension to the streamlet's output stream. The construct can be used to apply an operation to the elements of a multidimensional stream type while keeping the dimensionality information.

The signature of MapStream is:

$$Stream(T_{in}, d = D_{in}, c = C_{in}) \rightarrow Stream(T_{op}, d = D_{in}, c = C_{op})$$

T_{in} – Type encapsulated in the input stream.
D_{in} – The dimensionality of the input stream.
C_{in} – The complexity level of the input stream.
T_{op} – The type encapsulated in the output stream of the instantiated streamlet.
C_{op} – Instantiated module output stream complexity level.

The syntax of the MapStream pattern is:

```
map_inst: MapStream( op_inst: lib.Streamlet);
```

Nesting: The MapStream pattern can be nested to process streams with $D > 1$. In case of $D_{in} = 2, and D_{op} = 0$:

```
map_outer: MapStream( map_inner: MapStream(op_inst: lib.Streamlet));
```

Hardware design: The hardware architecture of the MapStream pattern is depicted in Figure 4.1. The design consists of two main components, the stream element counter, and the stream sequencer. These modules are responsible for managing the outermost dimension of the stream by ensuring that the same number of elements are handshaked on the input of a streamlet as on its output in the given dimension. The corresponding *last* signal is asserted whenever the last element is being handshaked on the output.

The element counter and sequencer modules are connected through a stream that transfers the length of the sequence whenever the last element appears on the input. The width of the stream equals to the system-wide index width, which is a constant representing the bitwidth of the indices.

Since the length of the sequence is unknown before the last element arrives, some of the elements most likely propagate through the user kernel before the sequence length arrives at the sequencer. The sequencer module can be implemented as a decrementing counter that counts the handshaked elements even before the next length value arrived. When the sequence length is known, it is added to the counter. When it reaches zero, it means the last element is being handshaked; hence, the last signal is asserted.

In order to accommodate kernels with any latency and sequences with any length, the sequencer module contains a FIFO for the incoming length values. This makes sure that the sequences shorter than the latency of the kernel are recreated correctly. The FIFO has to be sized in a way that it can hold the maximum number of length values that can occur within the maximum latency of the user streamlet, considering the shortest sequence length.

Additional consideration has to be taken to handle the case when the latency of the user streamlet is shorter than the latency of the element counter - sequencer path. If the latency is too low in the user streamlet path (e.g., a combinatorial kernel), short sequences can be missed. In order to ensure this cannot happen, the latency of the data path has to be adjusted to be longer than the latency between the element counter and the sequencer's counter. This is done by inserting stream slices into the data path. These can be elided in case the user streamlet has long enough latency.

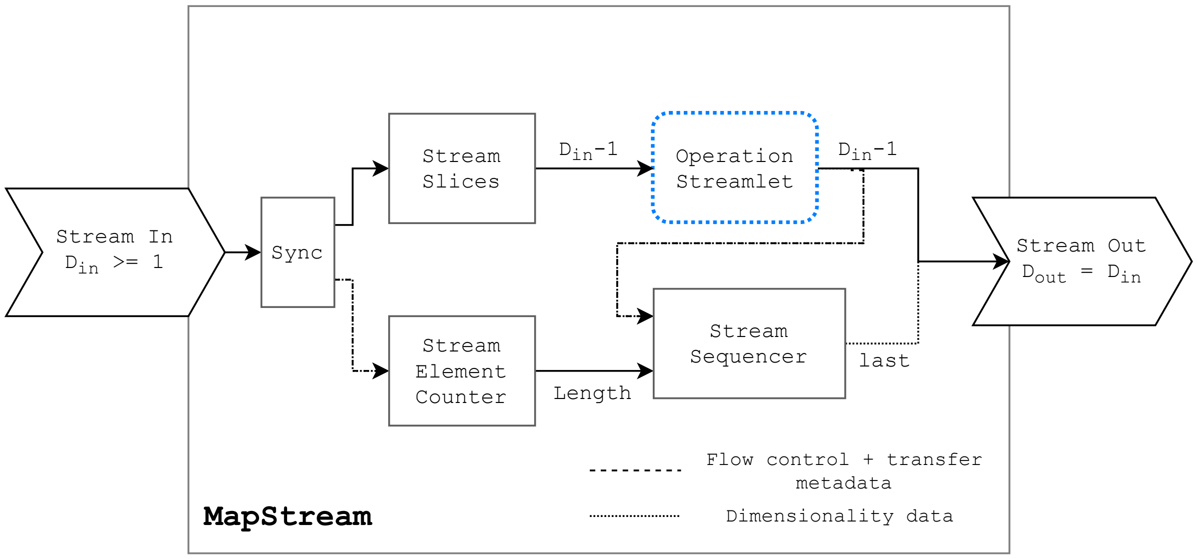


Figure 4.1: MapStream pattern hardware architecture.

FlatMapStream

The FlatMapPattern, similarly to MapStream, removes the outermost dimension, but instead of reconstructing the same dimensionality on the output, it discards it.

The signature of FlatMapStream is:

$$Stream(T_{in}, d = D_{in}, c = C_{in}) \rightarrow Stream(T_{op}, d = D_{in} - 1, c = C_{op})$$

where:

T_{in} – Type encapsulated in the input stream.
D_{in} – The dimensionality of the input stream.
C_{in} – The complexity level of the input stream.
T_{op} – The type encapsulated in the output stream of the instantiated streamlet.
C_{op} – Complexity level of the instantiated module's output stream.

The syntax of the FlatMapStream pattern is:

```
flatmap_inst: FlatMapStream( op_inst: lib.Streamlet);
```

Nesting: The FlatMapStream pattern can be nested to process streams with $D > 1$. In case of $D_{in} = 2$, and $D_{op} = 0$:

```
map_outer: FlatMapStream( map_inner: FlatMapStream(op_inst: lib.Streamlet));
```

ReduceStream

The ReduceStream pattern reduces a sequence of elements to a single value by collapsing the innermost dimension.

The signature of ReduceStream is:

$$Stream(T_{in}, d = D_{in}, c = C_{in}) \rightarrow Stream(T_{in}, d = D_{in} - 1, c = 1)$$

where:

T_{in} – Type encapsulated in the input stream.
D_{in} – The dimensionality of the input stream.
C_{op} – Complexity level of the instantiated module's output stream.

The syntax of the ReduceStream pattern is:

```
reduce_inst: ReduceStream(op_inst: lib.Streamlet);
```

Chaining: ReduceStream patterns can be chained to process streams with $D > 1$. In case of $D_{in} = 2$, and $D_{op} = 0$:

```
reduce_inner: ReduceStream(op_inst: lib.Streamlet);
reduce_outer: ReduceStream(op_inst: lib.Streamlet);
reduce_inner <=> reduce_outer;
```

Hardware design: The hardware architecture of the ReduceStream pattern is depicted in Figure 4.2. The architecture of said construct is similar to MapStream, with the difference being the included accumulator and the use of multiple counter-sequencer pairs. In case of ReduceStream, all the dimensions have to be managed, the outer dimensions to keep the $D_{in} - 1$ dimensionality, and the innermost to validate the output when the last element has been processed.

The streamlet used as the operator takes two operands. It is possible to exploit data parallelism by allowing the operand stream coming from outside to have multiple data lanes. This introduces a difference in counting compared to MapStream; the reduce operator doesn't necessarily produce the same number of output values as input values it consumed. Hence, in the case of the ReduceStream pattern, the number of transactions has to be counted instead of individual elements.

If the used complexity level allows it, empty transactions can appear. Since the operator is required to produce an equal amount of transactions on the input and the output, it's possible that the last transaction of the sequence is empty. In that case, the last accumulator value has to be sent, hence the multiplexer on the output.

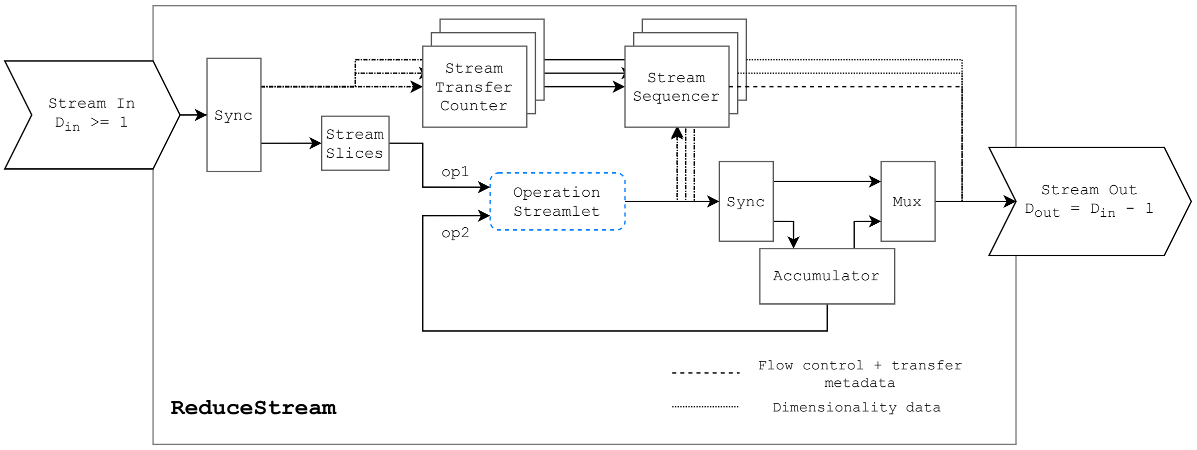


Figure 4.2: ReduceStream pattern hardware architecture.

FilterStream

The FilterStream construct takes a data and a predicate stream and filters the data stream with the predicates used as a mask.

The signature of FilterStream is:

$$Stream(T_{in}, d = D_{in}, t = t_{in}, c = C_{in}) \rightarrow Stream(T_{out}, d = D_{out}, t = t_{out}, c = 7)$$

where:

T_{in} – Type encapsulated in the input stream.
D_{in} – The dimensionality of the input stream.
C_{in} – The complexity level of the input stream.
T_{in} = T_{out}.
D_{in} = D_{out}.
t_{in} = t_{out}.

The syntax of the FilterStream pattern is:

```
filter_inst: FilterStream(inst.predicate);
```

Hardware design: The hardware design of the FilterStream pattern is depicted in Figure 4.3. The incoming predicates are stored in a FIFO and used as a mask for determining the validity bit on the output for every lane; hence, the *throughput* parameter of the incoming data and predicate stream has to be equal.

The FilterStream primitive implements filtering by consuming the elements on the input stream that have *false* associated with them on the predicate stream and passing the ones with a *true* predicate to the output.

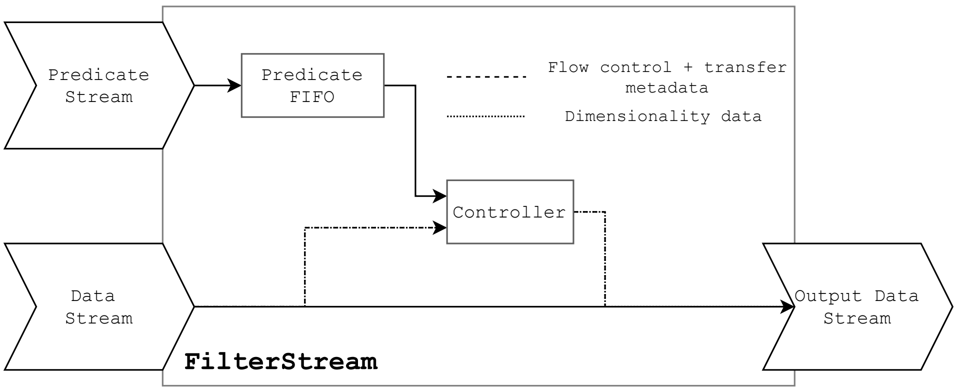


Figure 4.3: FilterStream pattern hardware architecture.

Handling dimensionality: The FilterStream construct keeps the dimensionality information of the incoming data stream. In case the last value of a sequence is associated with a *false* predicate, an empty transaction is sent on the output stream to signal the end of a sequence.

Output complexity level: The complexity level of the output stream is 7 if the input stream has a throughput parameter greater than one. In this case, the predicate values are used as a mask for determining the validity of the output lanes on the *strb* lanes. The level of complexity is explained by the case when the last element of a sequence arrives in a transaction that requires some of the lanes to be passed to the output. Given these conditions, any of the lanes can signal the end of a sequence on the output, which is the exact definition of complexity level 7. If the *throughput* parameter of the input cannot be matched by the sources, or the output complexity level is too high, gearbox and conversion logic should be inserted.

Nested data structures: The hardware architecture described above processes elementary streams. If the stream type consists of nested streams, additional sinks would be required to consume the nested elements. However, for filtering complex data structures with a considerable size, an index-based approach would be more suitable.

VectorToSeq

The VectorToSeq primitive takes a $Vector\langle T\rangle$ Tydi container and converts it to $Seq\langle T\rangle$.

Hardware design: The hardware architecture of the VectorToSeq primitive is depicted in Figure 4.4. The VectorToSeq primitive unwraps the $Vector$ container into the element and length stream. The length stream is forwarded to the stream sequencer that advances the output stream of the processing streamlet with a $last$ bit.

The syntax of the VectorToSeq primitive is:

```
seq_inst: VectorToSeq(inst.vec);
```

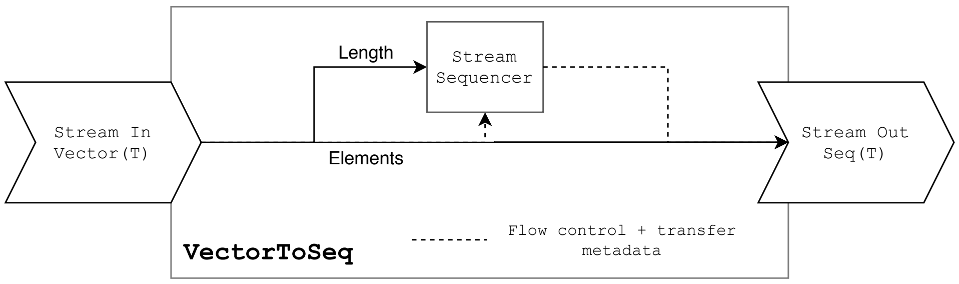


Figure 4.4: VectorToSeq pattern hardware architecture.

MapVector

The MapVector construct is analogous to MapStream, but instead of having a multidimensional stream as input, it takes a $Vector\langle S\rangle$ container.

In a $Vector\langle S\rangle$ container, the length of the sequence is indicated using a separate stream instead of increasing the dimensionality of the stream. Since the dimensionality information is separate from the data stream, and there are no *last* bits, counters are not required in this case.

The syntax of the MapVector pattern is:

```
mapvec_inst: MapVector( op_inst: lib.Streamlet);
```

Hardware design: The hardware architecture of the MapVector primitive is depicted in Figure 4.5. The MapVector primitive unwraps the $Vector\langle S\rangle$ container and feeds the element stream into the instantiated processing streamlet. The container is rebuilt on the output by passing the length stream through the primitive.

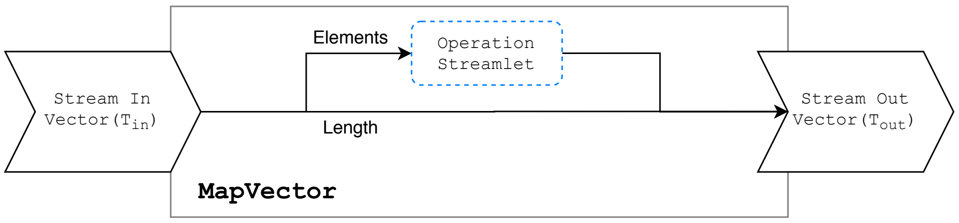


Figure 4.5: MapVector pattern hardware architecture.

ReduceVector

The ReduceVector is analogous to ReduceStream, but instead of having a multidimensional stream as input, it takes a $Vector\langle S\rangle$ container.

The syntax of the ReduceVector pattern is:

```
reduce_vectur_inst: ReduceVector( op_inst: lib.Streamlet);
```

Hardware design: The hardware architecture of the ReduceVector primitive is depicted in Figure 4.6. The overview of the design is similar to ReduceStream with the difference being that the vector length is known in advance; therefore, the element counter on the input stream is not required.

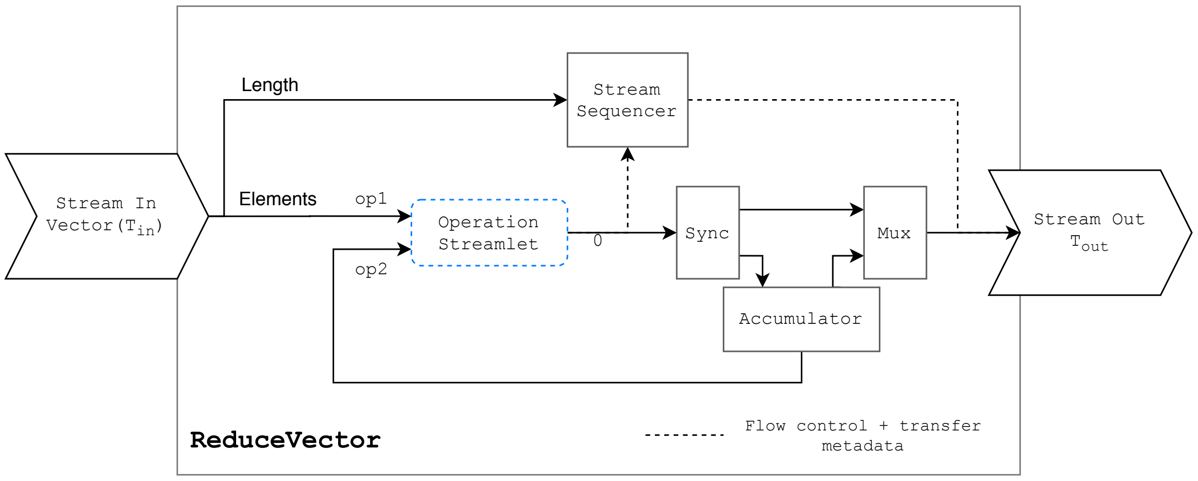


Figure 4.6: ReduceVector pattern hardware architecture.

4.1.3. Builders

The following constructs are hardware builders for container types. These primitives can be used to combine streams originating from multiple sources to a single interface.

ConcatStructBuilder The ConcatStructBuilder construct builds a streamlet that has an output interface of type $ConcatStruct\langle S_1, S_2, ..., S_n\rangle$, where $S_1, S_2, ..., S_n$ are the streamspace types of the input interfaces.

```
concat_struct_builder: ConcatStructBuilder(field_a <= inst1.a,
    field_b <= inst2.a);
```

DesyncStructBuilder The DesyncStructBuilder construct builds a streamlet that has an output interface of type $DesyncStruct\langle S_1, S_2, ..., S_n\rangle$, where $S_1, S_2, ..., S_n$ are the streamspace types of the input interfaces.

```
desync_struct_builder: DesyncStructBuilder(field_a <= inst1.a,
    field_b <= inst2.a);
```

PackedVariantBuilder The PackedVariantBuilder construct builds a streamlet that has an output interface of type $PackedVariant\langle S_1, S_2, ..., S_n\rangle$, where $S_1, S_2, ..., S_n$ are the streamspace types of the input interfaces.

```
packed_variant_builder: PackedVariantBuilder(field_a <= inst1.a,
  field_b <= inst2.a);
```

ConcatVariantBuilder The ConcatVariantBuilder construct builds a streamlet that has an output interface of type $ConcatVariant\langle S_1, S_2, ..., S_n\rangle$, where $S_1, S_2, ..., S_n$ are the streamspace types of the input interfaces.

```
concat_variant_builder: ConcatVariantBuilder(field_a <= inst1.a,
    field_b <= inst2.a);
```

4.1.4. Clone, split, demux

The constructs introduced in this section are defined to unwrap container types.

CloneStream CloneStream takes a list of interface names and replicates the input stream on all of those. This construct is required to be able to feed a stream into multiple streamlets.

```
clone_stream: CloneStream(a, b, c);
```

SplitConcatStruct SplitConcatStruct has an input interface with a type of $ConcatStruct\langle S_1, S_2, ..., S_n\rangle$ and splits the containerized members into separate output interfaces.

```
split_concat_struct: SplitConcatStruct(inst.if);
```

SplitDesyncStruct SplitDesyncStruct has an input interface with a type of $DesyncStruct\langle S_1, S_2, ..., S_n\rangle$ and splits the containerized members into separate output interfaces.

```
split_desync_struct: SplitDesyncStruct(inst.if);
```

DemuxPackedVariant DemuxPackedVariant has an input interface with a type of $PackedVariant\langle S_1, S_2, ..., S_n\rangle$ and splits the containerized members into separate output interfaces.

```
demux_packed_variant: DemuxPackedVariant(inst.if);
```

DemuxConcatVariant DemuxConcatVariant has an input interface with a type of $ConcatVariant\langle S_1, S_2, ..., S_n\rangle$ and splits the containerized members into separate output interfaces.

```
demux_concat_variant: DemuxConcatVariant(inst.if);
```

4.1.5. Discussion

The proposed language constructs allow the composition of complex hardware designs from streamlets that implement interfaces adhering to the Tydi specification. In addition, it defines a variety of patterns to address the complexity associated with the rich type system of Tydi, including composing, and deconstructing compound data types, or processing multidimensional streams. These constructs help to remove the control logic introduced by the context from streamlet.

By separating the context from the processing streamlets, standard libraries can be built featuring common operations from the big data field. These operations can be combined in a productive way to offload parts of the processing pipelines to the FPGA that may benefit from acceleration.

Although we focused on manual kernel development so far, the proposed language is also a good candidate to be used as an intermediate representation for code generation.

4.2. Practical use-case

To show the potential of the described primitives and composition language, a proof-of-concept design has been built using the FilterStream and ReduceStream constructs.

The chosen use-case is a typical dataframe query operation on a tabular dataset. The dataset of choice is the Chicago Taxi Trips dataset [5], which consists of metadata about taxi trips that were reported since 2013 to the agency in Chicago, such as the identifier of the taxi, the company, trip duration, and trip distance. The chosen use-case is to filter for a given company and calculate the total trip duration on the whole dataset.

4.2.1. Application overview

The dataset comes as an 80+ GB CSV file. The dataset was converted into Parquet file format in addition to selecting the required columns (company name, trip seconds) using Apache Spark. The Parquet version was then converted into an Apache Arrow RecordBatch using the Arrow Python API. The final dataset consists of 147M records, stored in a 3.9GB Arrow RecordBatch.

The system-level architecture of the application is depicted in Figure 4.7. It is a specialized case of the general Fletcher workflow that was explained in Chapter 2.

The host-side application prepares the data to be fed to the accelerator in an Arrow RecordBatch with a schema that has a *UInt64* and a *String* ($List\langle char\rangle$) field. The Fletcher design is also generated from this schema.

The data exchange between the host application and the accelerator kernel happens over OpenCAPI, using the CAPI SNAP framework and the Fletcher runtime.

The host-side application used during the profiling was written in C++ using the Fletcher C++ API. It copies the prepared RecordBatch from disk into the system memory and measures the elapsed time for all the accelerator-related calls.

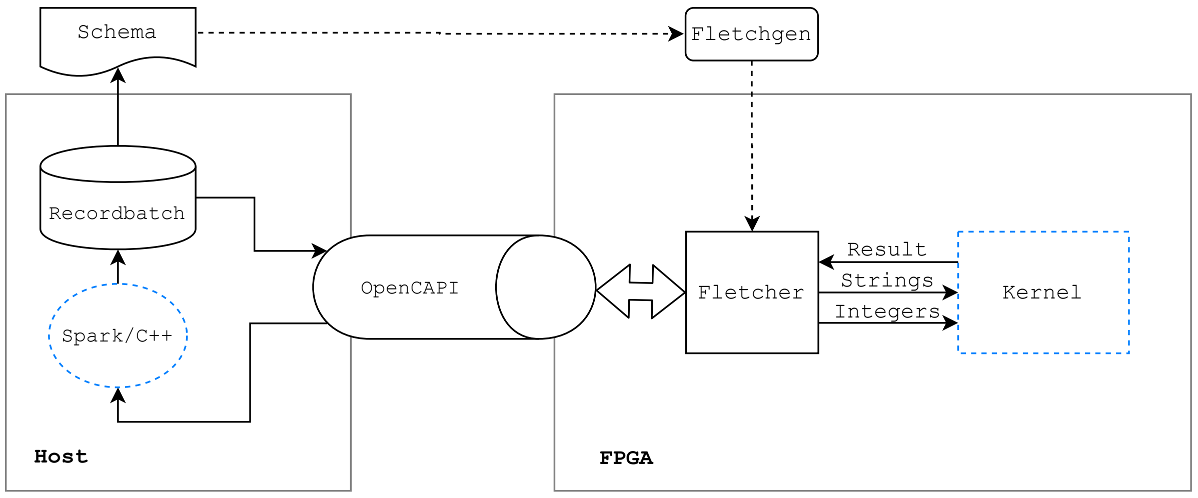


Figure 4.7: Example application top-level architecture.

4.2.2. Accelerator design

The high-level overview of the designed accelerator is shown in Figure 4.8. The kernel takes the two typed streams supplied by Fletcher. The stream carrying the company names as strings is connected to the regular expression matcher that is generated by the vhdre [10] tool. This module emits a boolean predicate stream for the filtering stage.

The output of the filter stage is fed into the reduce stage, which instantiates a sum operator and produces the output result. The output dimensionality of the stream is $D = 0$, since the records are represented as a single vector in the RecordBatch. The result is returned in an MMIO register to the host.

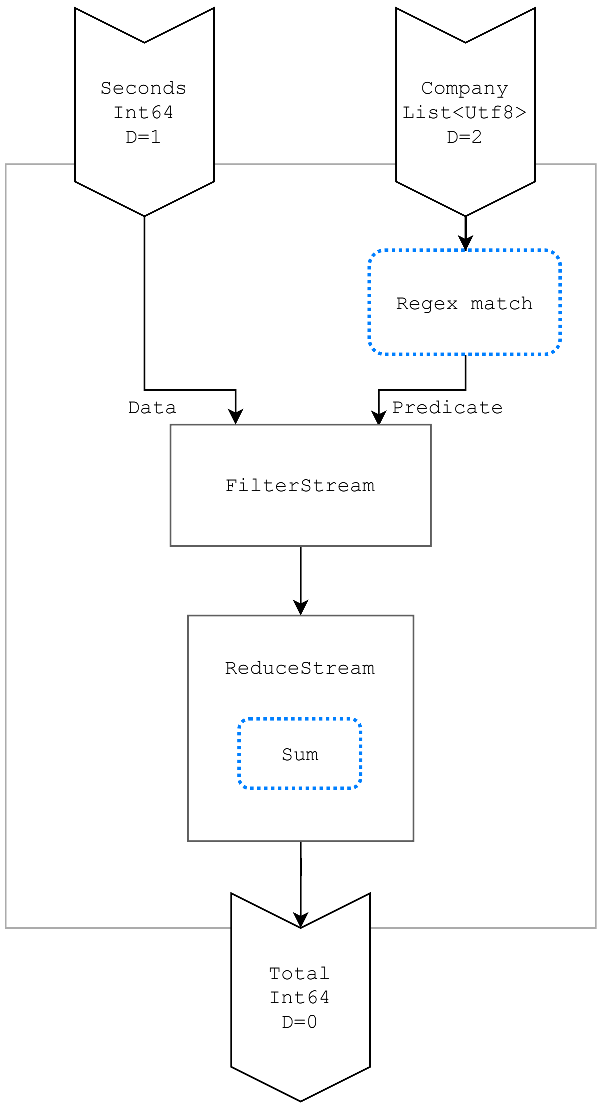


Figure 4.8: The architecture of the example design.

The representation of the design in the proposed composition language is shown in Figure 4.9. The streamlet definitions are separate from the structural implementation description. The definitions of the instantiated library components are shown in Figure 4.10. A Grahviz backend has been developed for Tydi that is capable of generating visualizations for the streamlets in a Tydi project, including their structural implementation. The output generated from the description above is shown in Figure 4.11. The dot diagram is the direct representation of the instantiated streamlets and the connection between their interfaces.

In this design there are 3 directly instantiated components: the regular expression matcher library component (*matcher*), the *FilterStream* pattern node (*filter_stage*), and the *ReduceStream* pattern node (*reduce_stage*). The input *chars* stream is fed into the regular expression matcher, which provides the predicate stream for the *FilterStream* node. The *chars* stream is capable of transferring 20 bytes per cycle. Since the strings in the *company name* field have a maximum of 40 characters, and limited to ASCII encoding, matching a single string takes at most 2 cycles. The *FilterStream* node filters input *numbers* stream and feeds it into the final *ReduceStream* node. The *numbers* stream only carries one element per cycle, since only one string is processed at a time. The output interface of the *Top_level* streamlet is the reduced stream.

If no basic operator library is assumed, the parts that would be up to the developer to write are the regular expression matcher and the sum operator streamlet.

```
impl compositions.Top_level structural {
  matcher: primitives.RegexMatcher[regex := ”Blue Ribbon Taxi Association Inc.”];
  matcher.in <= this.chars;
  filter_stage: FilterStream(matcher.out);
  reduce_stage: ReduceStream(reduce_op: primitives.Sum);
  filter_stage.in <= this.numbers;
  filter_stage <=> reduce_stage;
  this.out <= reduce_stage.out;
}
```

Figure 4.9: The composition code for the proof-of-concept accelerator.

```
Streamlet Sum (
    in: in Stream<Group<op1: Bits<64>, op2: Bits<64>>, d=0>,
    out: out Stream<Bits<64>, d=0>)
Streamlet RegexMatcher (
    in: in Stream<Bits<8>, t=20, d=1>,
    out: out Stream<Bits<1>, d=0>)
```

(a)

```
Streamlet Top_level (
    numbers: in Stream<Bits<64>, d=1>,
    chars: in Stream<Bits<8>, t=20, d=1>,
    out: out Stream<Bits<64>, d=0>)
```

(b)

Figure 4.10: (a) Streamlet definitions for the ”primitives” library. (b) Streamlet definition for the ”compositions” library.

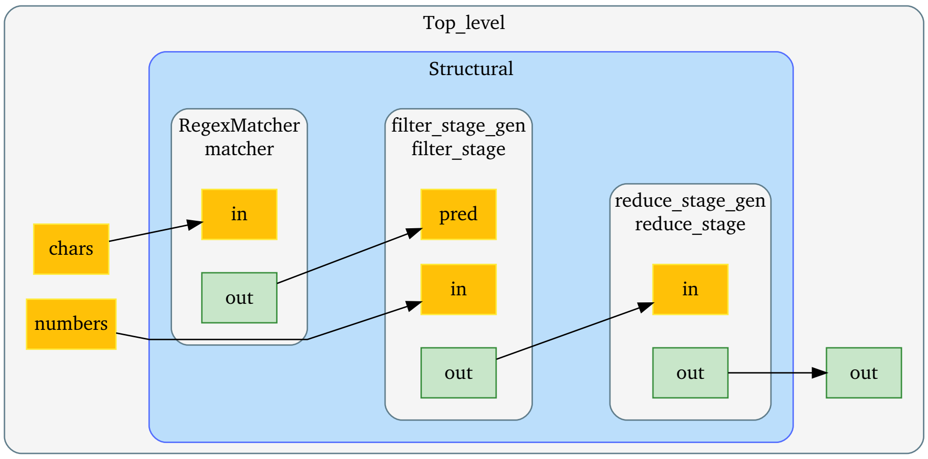


Figure 4.11: The architecture of the example design generated by the Graphviz backend.

5 Evaluation & results

The performance measurements of the proof-of-concept design were conducted on a Power9 system (PowerNV FP5290G2) with two CPUs (44 cores), and 128GB DDR4 memory. The FPGA card installed in the system is an AlphaData ADM-PCIE-9H7. The FPGA accelerator is interfaced with the system using OpenCAPI.

5.1. Code size comparison

The main offering of the structural composition language and Tydi is the productivity boost. To indicate the effort saved by this approach, we compare the code size of the final RTL design with the representation in the proposed language it can be generated from. The code size is measured without the comments and white spaces, and also excluding the reusable core components.

The code size of the final design is detailed in Table 5.1, and the composition code of the kernel is shown in Figure 4.9. It can bee seen that design can generated from 9 lines of composition code, while the composition-related part of the output codebase consists of 605 lines, which is a factor of 67 reduction in lines.

The code that has to be generated from the structural composition is mostly VHDL boilerplate with component instantiations and connections between streamlets. However, when patterns are instantiated, the code size difference becomes more significant.

Design partition	**Code size (lines)**
Fletcher	2499
Composed kernel	605
Operation (sum)	55
Regex matcher	550

Table 5.1: Code size composition of the example design.

5.2. FPGA utilization

The FPGA utilization of the complete design is shown in Table 5.2. The accelerator occupies 10.45% of the total FPGA capacity. The area distribution of the various components in the accelerator is further explained in Table 5.3. It can be seen that the user-kernel (*action* module) only accounts for 35.93% of the total design area, the the rest is the overhead introduced by the OC-Accel and CAPI SNAP framework hardware layers. These components are used to interface with OpenCAPI. The framework hardware consists of a AXI-to-CAPI bridge, memory-mapped register IO, and host DMA. It provides an AXI-Lite control interface, and coherent access to the system memory through AXI. These interfaces are used to integrate Fletcher as the user-action. The Fletcher infrastructure consists of a *Read arbiter* and the *RecordBatch reader*, which make up 2.73% and 27.5% of the area respectively. The share of the actual composed hardware (described in Section 4.2.2, *Kernel* module) is 4.86% of the whole design.

Resource	Used	Available	Utilization
CLB	17036	162960	10.45%
Block RAM tile	108	2016	5.33%

Table 5.2: FPGA utilization.

Module	Cells	%
FPGA	196929	100
└ BSP (OC-Accel Board Support Package)	62620	31.80
└ oc_func (OC-Accel + SNAP)	130525	66.28
└ action (SNAP user module wrapper)	70748	35.93
└ Fletcher AXI top	70709	35.91
└ Read arbiter (Fletcher)	5381	2.73
└ RecordBatch reader (Fletcher)	54147	27.50
└ Kernel Nucleus	11180	5.68
└ Kernel	9575	4.86
└ Regular expression matcher	4834	2.45
└ Filter stage	63	0.03
└ Reduce stage	597	0.3

Table 5.3: The area distribution of the relevant modules in the design.

5.3. Performance

The performance of the accelerator was measured using a purpose-built C++ application by directly profiling the relevant steps of the application execution. The results obtained with scaling the record count are shown in Figure 5.1. The total execution time is dominated by the ”enable context” step, which prepares the data and the accelerator for execution, and the ”waiting for kernel” step, which is the actual computation step executed on the accelerator. It can be seen that the runtime scales approximately linearly with the input size, as expected. From the results and the size of the dataset it can be calculated that the current implementation of the kernel is capable of processing $135M\ records/s$, which accounts to a throughput of $4.04GB/s$. However, taking into consideration the software overhead, mostly preparing the buffers, the performance drops to an average of $84M\ records/s$ with an average throughput of $2.51GB/s$.

If we consider that the kernel can consume $24\ bytes/cycle$ (character stream: $20\ bytes/cycle$, integer stream: $4\ bytes/cycle$), and that the kernel runs at $200Mhz$, the utilization of the streams during the execution is $\frac{4.04GB/s}{4.8GB/s} = 84\%$.

The dips in the graph need explaining: the buffers inside a RecordBatch are not necessarily aligned; however, CAPI SNAP expects 64-byte memory alignment. In order to ensure the required alignment, the buffers are reallocated whenever a misaligned buffer is queued using the Fletcher SNAP runtime. This reallocation results in a copy that degrades performance. In the case of the three dips that happened during the measurement, the character buffer happened to be aligned in the RecordBatch. By resolving the alignment issue, the overhead of preparing the data would be less significant.

Spark integration

Parallel to the development of the accelerator, F. Nonnenmacher explored the possibilities of accelerating Spark SQL [29], and the accelerator mentioned above was used as a proof-of-concept for the integration.

A measurement has been carried out with the same use-case, but using the Parquet input format and relying on Spark to orchestrate the process. The performance of the accelerated flow was compared to the single-core performance of the same application on a Power9 system. The results are shown in Figure 5.2. It can be seen that the execution time decreased to about half, with the Parquet reading step dominating the rest. The FPGA accelerated part of the computation shows a $13x$ speedup compared to the CPU version.

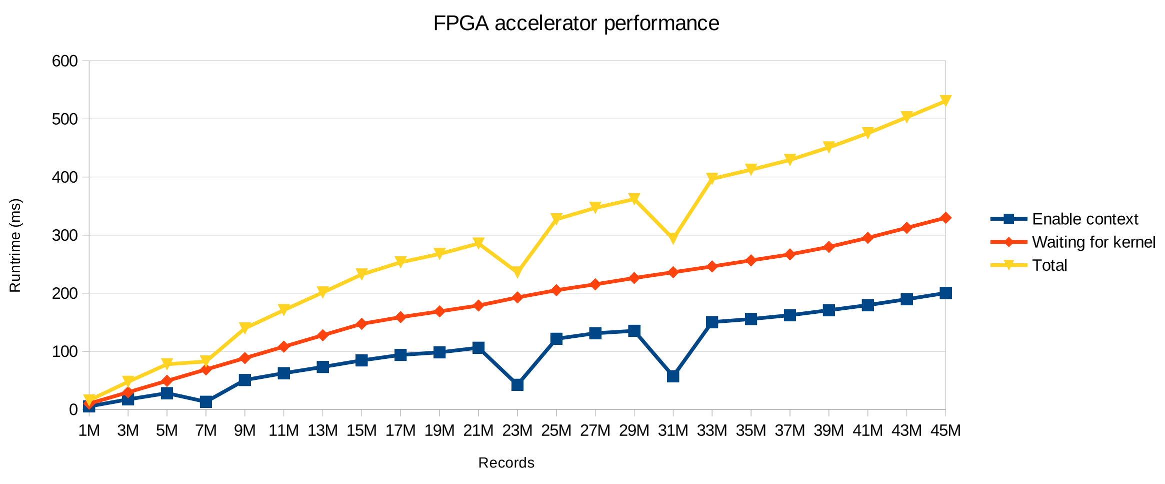


Figure 5.1: Proof-of-concept accelerator stand-alone performance.

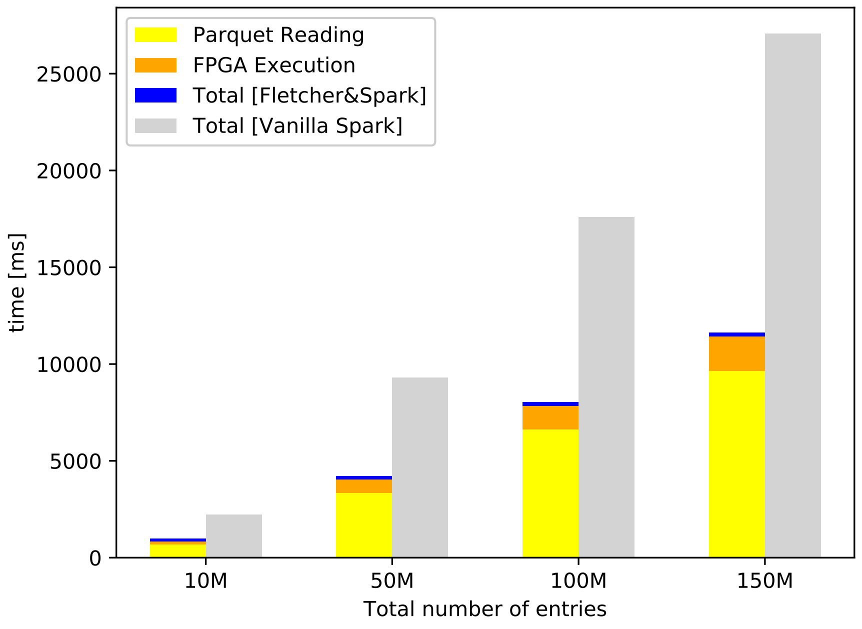


Figure 5.2: Proof-of-concept accelerator performance after integration. [29]

With the proposed composition language, an end-to-end workflow is complete for accelerating big data frameworks: Apache Arrow for data exchange, Fletcher for integration, and Tydi with the proposed composition language for describing the accelerator hardware for the offloaded operation.

The proposed approach is a good fit for a library-based acceleration workflow. A library of hardware accelerated functions can be built, and using the composition language the customized hardware can be assembled, either manually, or automatically by a code generator tool. The language constructs are suitable for describing computations present in the Spark dataflow graph, making it a straightforward representation for the offloaded functions.

5.4. Discussion

Since the FPGA utilization is only $\sim 10\%$, it would be possible to include more processing kernels. OpenCAPI is capable of transferring up to $12GB/s$ when a 512-bit AXI interface is used. In this case, 3 instances of the existing kernel would saturate the available bandwidth. Using a 1024-bit AXI interface, the bandwidth limit is $25GB/s$, for which 6 kernels would be required to saturate. Since only the kernel and the relevant parts of the Fletcher infrastructure would needed to be instantiated 6 times, that would still fit into the FPGA, considering that the kernel and the Fletcher infrastructure only accounts for 3.75% of the total FPGA capacity.

From a development point of view, it can be seen that the proposed language constructs indeed represent a productivity boost, and the higher-level, domain-specific constructs have adequate performance to be used in high-throughput accelerator development. The performance scaling is currently done using the *throughtput* parameter of the streams, which determines the numbers of elements that are transferable in a single cycle. Since the proposed parallel patterns constructs sit in the control path, they can handle the same throughput as the individual streamlet components.

6 Conclusions and recommendations

6.1. Conclusions

In this thesis, we aimed at providing a broad overview of the current digital design trends and identifying solutions that can be beneficially used in a big data context, either for manual development or automatic synthesis. A wide variety of solutions have been evaluated, ranging from modern HDLs to end-to-end solutions. It has been discovered that although there are frameworks available for productive accelerator development, there are limitations to their applicability in the envisioned workflow with Apache Arrow, Fletcher, and Tydi.

To address some of the shortcomings, a structural hardware composition language has been proposed for Tydi. The language includes constructs for defining implementations for streamlets, connecting streamlets, and a set of patterns for streamlining the development of domain-specific accelerators.

The productivity boost and utility of the proposed language has been demonstrated on a proof-of-concept design that represents the complete workflow of developing an accelerated application using Apache Arrow, Fletcher and Tydi.

The improvements made to Tydi during the thesis help productivity on two levels: the Chisel backend allows the developer to take advantage of a modern eDSL for streamlet development, and the composition language streamlines the development of dataflow designs using existing components. It has been shown on a real-world example that even for a simple design, the code-size reduction is close to two order of magnitude, and the accelerators built using the proposed constructs are performant to be used in large-scale acceleration. In particular, the sample accelerator that consisted of a filter and a reduce stage achieved a $13x$ speedup compared to the CPU implementation of the same processing task, while the kernel only occupies 3.5% of the FPGA.

Based on the findings of this thesis, we can answer the research questions in the following way:

1. *What are the current trends in digital hardware design which promise productivity improvement?*

   As the complexity of digital hardware increases with the available manufacturing technologies, new solutions are being developed to cope with this complexity on the design side. The main goal of the current state of the art is to move on from the RTL methodology to higher levels of abstraction. In the 2000s, high-level synthesis tools started to appear both in commercial and academic setups to approach hardware design from an algorithmic perspective. However, bridging the semantic gap between the sequential software constructs and the inherently parallel hardware has proven to be difficult. To this day, the main complaint about HLS tools is the detachment from the hardware. The control that designers have is at the algorithm-level and through constraints, not through direct specification. Recently, a new set of HLS tools is starting to gain traction, which synthesize hardware from algorithms described using implicitly parallel constructs: parallel patterns. These tools offer a convenient abstraction-level to describe algorithms that are the most prominent candidates for hardware acceleration due to their parallelism.

   In parallel, extensive research is going into developing new languages that can replace the RTL methodology, and offer hardware-centric abstractions. Some of these languages are Bluespec, Clash, TL-Verilog, Chisel, etc. Among the new hardware description languages, embedded domain-specific languages represent a considerable share. These eDSLs embed their hardware abstractions into a powerful host-language like Scala. This integration allows the developer to use the host language's constructs for hardware construction, and build abstractions as desired. The flexibility of this approach facilitates the engineering of workflows that otherwise may have been scattered around many tools and code generators.

2. *Which solutions are relevant to be used in a big data context?*

   ”Big data” is an umbrella term that covers a wide range of algorithms, applied in many research and commercial fields, and typically executed on large-scale computing platforms. This comprehensiveness leaves room for all of the discussed approaches. Some of the computations can be synthesized using HLS in a convenient and productive way, others may require hand-built and heavily optimized accelerators. In summary, we can say that the most promising solutions for big data are the frameworks that show some level of support for synthesizing hardware from parallel patterns, or if manual hardware development is required, one of the modern HDLs or eDSLs.

3. *Which solutions would be most suitable to be used a workflow with Apache Arrow and Fletcher?*

   Fletcher provides state of the art integration with big data frameworks through Apache Arrow. It interfaces with the in-memory, Arrow-formatted data using typed streams in hardware. For this reason, languages and frameworks that support streaming dataflow design are a natural fit. However, the custom streaming protocol set (Tydi) used by Fletcher has proven to be hard to accommodate in many of the existing solutions.

   To overcome these limitations, we proposed a composition language which beyond the basic structural description features offers a number of domain-specific constructs. Using these proposed constructs, the processing streamlets can be built using the most suitable technology, and combined together in a productive way.

6.2. Recommendations and future work

Buffer sizing: Buffers are inevitable components of dataflow design; however, the buffer sizes have to be chosen carefully to maximize performance and to prevent stalls. Since the proposed solution combines designs that can be seen as black-boxes, the latencies required for buffer sizing are not known. In order to tackle these issues, a solution is required to specify relevant metrics about the kernel, such as minimum, maximum, average latency, and initiation interval. From a semantics point of view, this would be a better fit for the streamlet specification rather than the composition language.

Generative constructs: Defining additional, more generative constructs would increase productivity even more. For example: *interface arrays* and *generative for loop*.

Behavioral constructs: In addition to the structural composition of streamlets, it would be desirable to implement behavioral constructs. The Fleet framework's eDSL and Spatial (introduced in Chapter 3) proposes language features that would be genuinely powerful combined with Tydi's rich, hardware-oriented type system.

Massively parallel streaming model: Currently, performance scaling is only possible by manipulating the throughput ratio property of the streams, or by instantiating multiple processing units by hand, provided that arbitration is feasible. A solution analogous to the one presented in Fleet [38] would be a valuable construct in the composition language to scale performance.

Formal verification properties for Tydi: Since Tydi aims to be an open specification for component interfaces, it would be desirable to have a formal and exhaustive way of verifying compliance with the specification for the given interface type and complexity level.

Using Tydi, complex designs can be composed, involving many individual components, where each component might come with its own testbench. However, ideally, integration tests should also be performed to verify the design as a whole. In the proposed workflow, this is done by testing on a sample

Arrow RecordBatch, or by using the target platform's simulator. This may not provide sufficient coverage to catch rare bugs.

Generating a set of formal checks and a verification harness for the streamlets in a Tydi project from the interface specifications would greatly reduce the risk of erroneous behavior, invalid transactions, and stalls caused by uncaught corner cases.

SymbiYosys [4] is an open source tool that could be used as a formal verification backend to check the formal properties.

Modern IR backend: Nowadays, Verilog and VHDL are only used as an intermediate representation by many of the higher-level frameworks to generate the final hardware description to be built using the toolstack of the hardware target. It would be a leap forward if Tydi were to use one of the emerging intermediate representations as a backend. LLHD [35] is one of the promising frameworks. LLHD has 3 levels to capture all aspects of digital design: *Behavioral LLHD*, *Structural LLHD*, and *Netlist LLHD*. LLHD could be used to generate the composition language constructs (Chapter 4), the instantiations of the streamlets, the connections between them, and to include verification assertions with the design to be used in simulation or for formal verification. Unfortunately, at the time of the writing, the tools used in high-performance acceleration do not support LLHD, and LLHD does not have a backend that could generate Verilog/VHDL that could be synthesized using the commercial tools.

FIRRTL [23] is a similar project that is used by Chisel to perform circuit transformations and emit Verilog for legacy tools.

Throughput estimation: Based on the throughput ratio and by introducing an average throughput property to the streamlets it would be possible to estimate the throughput of the composed design and find the bottlenecks by traversing the implementation graph. This would help during development to balance critical paths.

Design space exploration Once a wide set of performance-related features is incorporated, such as buffer sizing and massively parallel streaming model, design space exploration could be performed using the introduced parameters to find a suitable combination.

A
PEST grammar

```
// Whitespace and comments.
WHITESPACE              = _{ (" " | "\n") }
COMMENT                 = _{ "/*" ~ (!"*/" ~ ANY)* ~ "*/" }

// Identifiers
ident                   = @{(ASCII_ALPHANUMERIC | "_")+}

// Literals
number                  = @{ "-"? ~ int ~ ("." ~ ASCII_DIGIT+ ~ exp? | exp)? }
int                     = @{ "0" | ASCII_NONZERO_DIGIT ~ ASCII_DIGIT* }
exp                     = @{ ("E" | "e") ~ ("+" | "-")? ~ ASCII_DIGIT+ }

bool                    = {"true" | "false" }

char                    = {
                        !("\"" | "\\") ~ ANY
                        | "\\" ~ ("\"" | "\\" | "/" | "b" | "f" | "n" | "r" | "t")
                        | "\\" ~ ("u" ~ ASCII_HEX_DIGIT{4})
                        }


inner                   = @{ char* }
string                  = ${ "\"" ~ inner ~ "\"" }

node_if_handle          = { (ident ~ "." ~ ident) }
streamlet_handle        = { (ident ~ "." ~ ident) }

//Assign parameter
parameter_assign        = { ident ~ ":=" ~ (ident | string | number)}

//Parallel patterns
map_stream              = { "MapStream" ~ "(" ~ node ~ ")" }
filter_stream           = { "FilterStream" ~ "(" ~ node_if_handle ~ ")" }
reduce_stream           = { "ReduceStream" ~ "(" ~ node ~ ")" }
map_vector              = { "MapVector" ~ "(" ~ node ~ ")" }
reduce_vector           = { "ReduceVector" ~ "(" ~ node ~ ")" }
vector_to_seq           = { "VectorToSeq" ~ "(" ~ node_if_handle ~ ")" }
pattern                 = { map_stream | filter_stream | reduce_stream
                            | map_vector | reduce_vector | vector_to_seq}


concat_struct_builder   = { "ConcatStructBuilder(" ~ connection_in_place + ~ ")" }
desync_struct_builder   = { "DesyncStructBuilder(" ~ connection_in_place + ~ ")" }
```

```

concat_variant_builder  = { ”PackedVariantBuilder(” ~ connection_in_place + ~ ”)”
    }
packed_variant_builder  = { ”PackedVariantBuilder(” ~ connection_in_place + ~ ”)”
    }

builder                 = { concat_struct_builder | desync_struct_builder
                          | concat_variant_builder | packed_variant_builder}


split_concat_struct     = { ”SplitConcatStruct” ~ ”(” ~ node_if_handle ~ ”)” }
split_desync_struct     = { ”SplitDesyncStruct” ~ ”(” ~ node_if_handle ~ ”)” }
demux_packed_variant    = { ”DemuxPackedVariant” ~ ”(” ~ node_if_handle ~ ”)” }
demux_concat_variant    = { ”DemuxConcatVariant” ~ ”(” ~ node_if_handle ~ ”)” }

unwrap                  = { split_concat_struct |  split_desync_struct
                          | demux_packed_variant | demux_concat_variant}


clone_stream            = { ”CloneStream” ~ ”(” ~ ident+ ~ ”)” }

//Single point-to-point connection
connection              = { node_if_handle ~ ”<=” ~  node_if_handle }
connection_in_place     = { ident ~ ”<=” ~  node_if_handle }

//Chain connection
chain_connection        = { (ident) ~ (”<=>” ~ (ident))+ }

//Streamlet instantiation
streamlet_inst          = { streamlet_handle ~ (”[” ~ (parameter_assign)+ ~ ”]”)?
    }

//A node in the implementation graph
node                    = { ident ~ ”:” ~  (pattern | builder | unwrap |
    streamlet_inst) }

//Implementation of a streamlet
structural_body         = { (( connection | chain_connection | node) ~ ”;”)* }
structural              = { ”structural” ~ ”{” ~ structural_body ~ ”}”}
hdl                     = { ”VHDL” | ”Chisel” }
external                = { ”external” ~ string}
implementation          = { ”impl” ~ streamlet_handle ~ (structural | hdl |
    external) }
```

Bibliography

[1] Apache Arrow website. URL `https://arrow.apache.org/`.

[2] Fletcher Github. URL `https://github.com/abs-tudelft/fletcher`.

[3] PEST parser. URL `https://pest.rs/`.

[4] SymbiYosys Github repository. URL `https://github.com/YosysHQ/SymbiYosys`.

[5] Chicago Taxi Trips dataset. URL `https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew`.

[6] Cocotb, a coroutine based cosimulation library for writing VHDL and Verilog testbenches in Python. URL `https://github.com/cocotb/cocotb`.

[7] Fleet framework Github repository. URL `https://github.com/jjthomas/Fleet`.

[8] Tydi: an open specification for complex data structures over hardware streams. URL `https://abs-tudelft.github.io/tydi`.

[9] Verilator. URL `https://www.veripool.org/wiki/verilator`.

[10] vhdre: a VHDL regex matcher generator. URL `https://github.com/abs-tudelft/vhdre`.

[11] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 303–315, 2014. ISSN 1089795X. doi: 10.1145/2628071.2628092.

[12] Ayupov Andrey and Steven Burns. Chisel based HW design. URL `https://github.com/intel/rapid-design-methods-for-developing-hardware-accelerators/wiki/Chisel-based-HW-design`.

[13] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. *Proceedings - Design Automation Conference*, pages 1216–1225, 2012. ISSN 0738100X. doi: 10.1145/2228360.2228584.

[14] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Stephen Heil, Joo Young Kim, Daniel Lo, Michael Papamichael, Todd Massengill, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. *IEEE Micro*, 2017. ISSN 19374143. doi: 10.1109/MM.2017.265085811.

[15] Henry Cook, Wesley Terpstra, and Yunsup Lee. Diplomatic Design Patterns: A TileLink Case Study. *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters by. *Communications of the ACM*, 51(1):107–113, 2008. ISSN 00010782. doi: 10.1145/1327452.1327492. URL `http://www.usenix.org/events/osdi04/tech/full{_}papers/dean/dean{_}html/`.

[17] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H. Peter Hofstee. Refine and recycle: A method to increase decompression parallelism. *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, 2019-July:272–280, 2019. ISSN 10636862. doi: 10.1109/ASAP.2019.00017.

[18] Steven Hoover. Timing-Abstract Circuit Design in Transaction-Level Verilog. (Iccd):525–532, 2017.

[19] Steven Hoover and Ahmed Salman. Top-Down Transaction-Level Design with TL-Verilog.

[20] Joost Hoozemans, Rolf Heij, Jeroen van Straten, and Zaid Al-Ars. VLIW-Based FPGA Computation Fabric with Streaming Memory Hierarchy for Medical Imaging Applications. In Stephan Wong, Antonio Carlos Beck, Koen Bertels, and Luigi Carro, editors, *Applied Reconfigurable Computing*, pages 36–43, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56258-2.

[21] Ernst Joachim Houtgast, Vlad Mihai Sima, Koen Bertels, and Zaid Al-Ars. Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths. *Computational Biology and Chemistry*, 75:54–64, 2018. ISSN 14769271. doi: 10.1016/j.compbiolchem.2018.03.024. URL `https://doi.org/10.1016/j.compbiolchem.2018.03.024`.

[22] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support to Blaze FPGA accelerator deployment at datacenter scale. *Proceedings of the 7th ACM Symposium on Cloud Computing, SoCC 2016*, pages 456–469, 2016. doi: 10.1145/2987550.2987569.

[23] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2017-Novem:209–216, 2017. ISSN 10923152. doi: 10.1109/ICCAD.2017.8203780.

[24] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. FireSim: FPGA-Accelerated cycle-Exact scale-Out system simulation in the public cloud. *Proceedings - International Symposium on Computer Architecture*, pages 29–42, 2018. ISSN 10636897. doi: 10.1109/ISCA.2018.00014.

[25] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, pages 115–127, 2016. doi: 10.1109/ISCA.2016.20.

[26] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–311, 2018. ISSN 0362-1340. doi: 10.1145/3192366.3192379.

[27] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. High-level synthesis of functional patterns with lift. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–45, 2019. doi: 10.1145/3315454.3329957.

[28] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys*, 52(6), 2019. ISSN 15577341. doi: 10.1145/3357375.

[29] Fabian Nonnenmacher. Transparently Accelerating Spark SQL Code on Computing Hardware.

[30] Johan Peltenburg, Shanshan Ren, and Zaid Al-Ars. Maximizing systolic array efficiency to accelerate the PairHMM Forward Algorithm. *Proceedings - 2016 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2016*, pages 758–762, 2017. doi: 10.1109/BIBM.2016.7822616.

[31] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H Peter Hofstee, and Zaid Al-Ars. Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow. In Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz, editors, *Applied Reconfigurable Computing*, pages 32–47, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17227-5.

[32] Johan Peltenburg, Jeroen Van Straten, Lars Wijtemans, Lars Van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate FPGA accelerators with apache arrow. *Proceedings - 29th International Conference on Field-Programmable Logic and Applications, FPL 2019*, pages 270–277, 2019. doi: 10.1109/FPL.2019.00051.

[33] Johannus Willem Peltenburg, Matthijs Brobbel, Jeroen Van Straten, Zaid Al-Ars, and Peter Hofstee. Tydi: an open specification for complex data structures over hardware streams. *IEEE Micro*, pages 1–10, 2020. ISSN 19374143. doi: 10.1109/MM.2020.2996373.

[34] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, Hyoukjoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 02-06-Apri:651–665, 2016. doi: 10.1145/2872362.2872415.

[35] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. LLHD: a multi-level intermediate representation for hardware description languages. pages 258–271, 2020. doi: 10.1145/3385412.3386024.

[36] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce framework on FPGA a case study of RankBoost acceleration. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*, (May 2014):93–102, 2010. doi: 10.1145/1723112.1723129.

[37] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. LIFT: A functional data-parallel IR for high-performance GPU code generation. *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 74–85, 2017. doi: 10.1109/CGO.2017.7863730.

[38] James Thomas, Pat Hanrahan, and Matei Zaharia. Fleet: A framework for massively parallel streaming on FPGAS. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 639–651, 2020. doi: 10.1145/3373376.3378495.

[39] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. *Leibniz International Proceedings in Informatics, LIPIcs*, 136(7):1–7, 2019. ISSN 18688969. doi: 10.4230/LIPIcs.SNAPL.2019.7.

[40] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. Melia: A MapReduce Framework on OpenCL-Based FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, 2016. ISSN 10459219. doi: 10.1109/TPDS.2016.2537805.

[41] Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Andrey Ayupov, Steven Burns, and Ozcan Ozturk. Hardware accelerator design for data centers. *2015 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015*, pages 770–775, 2016. doi: 10.1109/ICCAD.2015.7372648.

[42] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. *Proceedings - Design Automation Conference*, Part F1377:0–5, 2018. ISSN 0738100X. doi: 10.1145/3195970.3196109.