Methods for Efficient Integration of FPGA Accelerators with Big Data Systems

Peltenburg, J.W.

DOI
10.4233/uuid:51989f8f-f672-4f4b-a059-86233869ff47

Publication date
2020

Document Version
Final published version

Citation (APA)
Peltenburg, J. W. (2020). *Methods for Efficient Integration of FPGA Accelerators with Big Data Systems*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:51989f8f-f672-4f4b-a059-86233869ff47

Methods for Efficient Integration of FPGA Accelerators with Big Data Systems

Johan Peltenburg

Methods for Efficient Integration of FPGA Accelerators with Big Data Systems

METHODS FOR EFFICIENT INTEGRATION OF FPGA ACCELERATORS WITH BIG DATA SYSTEMS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 3 november 2020 om 15:00 uur

door

Johannus Willem PELTENBURG

Master of Science in Computer Engineering,
Technische Universiteit Delft,
geboren te Middelharnis, Nederland.

Dit proefschrift is goedgekeurd door de promotoren.

Samenstelling promotiecommissie bestaat uit:

Rector Magnificus,	voorzitter
Prof. dr. H.P. Hofstee,	Technische Universiteit Delft, Promotor
Dr. ir. Z. Al-Ars,	Technische Universiteit Delft, Promotor
Onafhankelijke leden:	
Prof.dr.ir. W.A. Serdijn	Technische Universiteit Delft
Prof.dr.ir. K.L.M. Bertels	Technische Universiteit Delft
Prof.dr. J.H. Takala	Tampere University of Technology
Prof. Dr.-Ing. Dr. h.c. J. Becker	Karlsruhe Institute of Technology
Overige leden:	
Dr. J.S. Rellermeyer	Technische Universiteit Delft

Keywords: Big Data, FPGA, accelerators

Printed by: Gildeprint

Front & Back: (CC BY-SA) J.W. Peltenburg

ISBN 978-94-6366-333-5

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

Contents

Summary

Because of fundamental limitations of CMOS technology, computing researchers and the computing industry are focusing on using transistors in integrated circuits more efficiently towards obtaining a computational goal. At the architectural level, this has led to an era of heterogeneous computing, where various types of computational components are used to solve problems. In this dissertation, we focus on the integration of one such heterogeneous component; the FPGA accelerator, with one of the main drivers behind the increasing need of computational performance; big data systems. With the increased availability of these FPGA accelerators in data centers and clouds, and with an increasing amount of I/O bandwidth between accelerated systems and their host, the industry is trying to push these components into more widespread usage in big data applications. For big data systems, three related challenges are observed. First, the software systems consist of many layered run-time systems that have often been designed to raise the level of abstraction, often at the cost of potential performance. Second, hardware-unfriendly in-memory data structures, and (to the accelerator) uninteresting metadata may convolute designs required to integrate FPGA accelerators with big data systems software. Last, serialization is applied to face the second challenge, but the rate at which serialization is performed is much lower than the rate at which accelerators may absorb data. For FPGA accelerators, we also observe three challenges. First, highly vendor-specific styles of designing hardware accelerators hampers the widespread reuse of existing solutions. Second, developers spend a lot of time on designing interfaces appropriate for their data structure, since they are typically provided with just a byte-addressable memory interface. Third, developers spend a lot of time on the infrastructure or 'plumbing' around their computational kernels, while their focus should be the kernel itself. We describe a toolchain named Fletcher, based on the Apache Arrow in-memory format for tabular data structures, that uses Arrow to deal with the challenges on the big data systems software side, and also deals with the challenges on the FPGA accelerator development side. The toolchain allows to rapidly generate platform-agnostic FPGA accelerator designs where kernels operate on tabular data sets, requiring the developer to only implement the kernel, automating all other aspects of the design, including hardware interfaces, hardware infrastructure, and software integration. We describe applications in regular expression matching, k-means clustering, Hidden Markov Models with the posit numeric format, and decoding Parquet files. We finally apply the lessons learned on the work of the Fletcher framework in a new interface specification for streaming dataflow designs, named Tydi. We introduce a hardware-oriented type system that allows to express complex, dynamically sized data structures often found in the domain of big data analytics. The type system helps to increase the productivity when designing hardware transporting such data structures over streams, abstracting their use in hardware without losing the ability to make common design trade-offs.

Samenvatting

Omdat fundamentele limieten van CMOS technologie in zicht zijn, richten onderzoekers naar computertechniek en de computerindustrie zich op het efficiënter gebruiken van transistoren in geïntegreerde circuits, zodanig dat de transistoren beter gebruikt worden om het rekenkundige doel te bereiken. Op het niveau van de architectuur heeft dit geleid tot het tijdperk van heterogene computers, waarbij verschillende soorten componenten gebruikt worden om problemen op te lossen. In deze uiteenzetting beschouwen we de integratie van één soort heterogeen onderdeel; de FPGA-versneller, met één van de grootste afnemers van computercapaciteit; big data systemen. Nu dat meer van deze FPGA-versnellers beschikbaar zijn in datacentrums en *clouds*, en met een stijgende hoeveelheid invoer- en uitvoerbandbreedte tussen versnelleronderdelen en hun gastheer, lijkt het erop dat de industrie probeert deze onderdelen beschikbaar te maken voor een breder publiek dat werkt aan *big data* toepassingen. Aan de kant van big data systemen observeren we drie uitdagingen. Ten eerste bestaan de softwaresystemen uit vele lagen die ontworpen zijn om het niveau van abstractie te verhogen, vaak ten koste van prestatievermogen. Ten tweede, door hardware-onvriendelijke opmaak van datastructuren in het geheugen, en door (voor de versneller) oninteressante metadata wordt het ontwerpen van FPGA versnellers die geïntegreerd zijn met de software van big data systemen bemoeilijkt. Als laatste wordt *serialisatie* vaak toegepast om de voorgaande uitdaging aan te gaan, maar de snelheid waarmee serialisatie plaats kan vinden is veel lager dan de snelheid waarmee versnellers data kunnen absorberen. Aan de kant van FPGA-versnellers observeren we ook drie uitdagingen. Ten eerste is de ontwikkelstijl vaak in hoge mate toegespitst op specifieke technieken van bedrijven, wat het hergebruik van bestaande oplossingen tegenhoudt. Ten tweede spenderen ontwikkelaars veel tijd aan het ontwerpen van de juiste koppelstukken die overeenkomen met hun datastructuren, omdat ze typisch alleen bytegeaddresseerde geheugenkoppelstukken aangeboden krijgen. Ten derde spenderen ontwikkelaars veel tijd aan het ontwerpen van de infrastructuur of 'pijpwerk' rondom de rekenkundige kernen, terwijl hun focus moet liggen op de rekenkundige kernen zelf. We beschrijven een verzameling van gereedschappen, genaamd Fletcher, gebaseerd op de opmaak voor tabulaire datastructuren van het Apache Arrow project. Hierbij gaat Arrow de uitdagingen met betrekking tot de big data systemen aan, en Fletcher de uitdagingen met betrekking tot de FPGA-versnellers aan. De verzameling van gereedschappen staat toe om snel platformonafhankelijke ontwerpen te genereren voor FPGA-versnellers, waarbij kernen opereren op tabulaire datasets. Hierdoor hoeft de ontwikkelaar alleen de kernen te ontwerpen, en worden alle overige aspecten van het ontwerp geautomatiseerd, inclusief de hardwarematige koppelstukken, hardwarematige infrastructuur en software-integratie. We beschrijven toepassingen in reguliere uitdrukkingen, het algorithme van k-gemiddelden, verborgen Markov modellen met het *posit* nummerformaat, en het ontleden van Parquet-bestanden. Als laatste passen we de lessen die geleerd zijn tijdens het werken aan Fletcher toe in een nieuwe koppelstukspecificatie voor stromende data-ontwerpen, genaamd Tydi. We introduceren hierbij een typesysteem georiënteerd op hardware, dat toestaat om complexe datastructuren van dynamische afmetingen uit te drukken, zoals deze vaak in het domein van big data analyse te vinden zijn. Het typesysteem helpt met het verhogen van de productiviteit bij het ontwerpen van hardware waarbij zulke datastructuren over stromen vloeien. Hierbij wordt het niveau van abstractie verhoogd zonder de mogelijkheid te verliezen om de gebruikelijke afwegingen te maken in het ontwerp.

1

INTRODUCTION

It was certainly rather attractive, and though he was wisely cautious of most new things, he did not hesitate for long before sidling up to it.

Arthur C. Clarke, from “2001: A Space Odyssey”

This thesis describes methods to efficiently integrate FPGA accelerators with contemporary software systems found in the domain of big data analytics. The topic encompasses a very broad set of related technologies that are to be taken into consideration. In this chapter, we will give a historical perspective on the technologies to explain how the desire to combine them came to be. We briefly touch on general-purpose processors, heterogeneous computing, field-programmable gate arrays, high-performance computing, and big data analytics frameworks. We then continue to describe problems in this context, and pose several research questions that this thesis aims to answer, with the main question being: how can FPGA accelerators be integrated efficiently with contemporary big data systems software?

1.1. A PERSPECTIVE ON COMPUTING

1.1.1. DIGITAL INFORMATION IN SOCIETY

PRESENT-DAY society is ever more dependent on information technology. Before the Digital Revolution that brought humanity into the Information Age, it was once hard to imagine that computers would have a place in every household, let alone that not just every family would have one, but that every *person* would have *several* — sometimes even *carrying them* on or *inside their bodies*! It is easier than ever to gather and distribute digital information, due to increased connectivity and computational capability of these computing devices of all shapes and sizes used by people, companies, institutions and governments.

Many decisions in society, politics, companies, and the daily lives of humans are increasingly based on the result of analysis of very large amounts of shared and stored digital information. The success stories of the so called field of *big data* [1] are many.

Outcomes of analyzing big data influence our daily lives in a positive manner. For example, through analysis of large DNA databases, a better understanding of human and plant diseases is made possible, allowing us to develop new medicines, design new treatments, and rapidly breed resilient crops. Machine learning systems trained on terabytes of images can now detect specific tumors based on just a single image, with higher precision and do so much faster than experienced surgeons. Petabytes of sensory data from around the globe help to understand major challenges of the new century, such as climate change. Hundreds of petabytes of sensory data from the Large Hadron Collider help us understand the fundamentals of our universe.

More worrisome outcomes exist as well. Privacy-invading technologies to construct extensive psychological profiles of billions of potential customers browsing the internet results in increased brand exposure and company revenue through targeted advertisements. The analysis of thousands of camera feeds with face recognition tracks the everyday movement of unsuspecting citizens, and allows building up automated population control systems. Mass surveillance and spreading of digital misinformation to influence the outcomes of elections in rivaling countries headlines news bulletins every day. The famous saying *scientia potentia est* (knowledge is power) easily comes to mind.

1.1.2. TIME IS OF THE ESSENCE

Analyzing large volumes of digital information requires computers. It is not surprising that those that design solutions (software and hardware *developers*) to big data problems want these computers to be both fast and easy to use. After all, within a given budget, it is the total time spent to come up with a solution that often matters most. The sooner a new medicine is developed, the sooner people can start getting cured. The sooner psychological profiles of potential customers are analysed, the sooner an advertisement company will have the competitive edge.

Often very consciously (but sometimes rather unconsciously), an application developer therefore attempts to minimize the time to solution; they prefer the most productive approach where the solution to their problem appears as fast as possible, within their budget. But what are the factors that contribute to this total time spent on solving a computational problem? This question is not easy to answer, since the parameters that

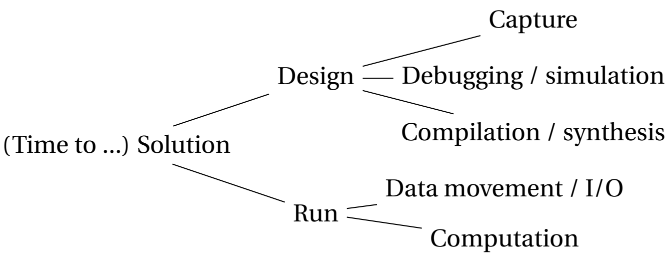


Figure 1.1: Breakdown of time to solution

contribute to it are virtually infinite, especially if the political, economic and human dimensions are taken into account. Considering the question from a more technical perspective, a very high-level breakdown is less difficult to construct, as seen in Figure 1.1. Here, we discern two major components; the design time and the run time.

The *design time* is the time spent on describing a solution that connects and instructs our computational platform in such a way that it solves a problem; e.g. writing a computer program or designing a circuit. The *run time* is the time spent by the computer to solve the problem according to the design.

We may break the design time up into the time used to *capture* the description of the solution, fix human errors in the description through *debugging or simulation*, and finally *synthesize or compile* the often abstracted description into something that physically maps onto a specific computing platform. The run time may be broken up into the time spent on *moving the data* from wherever they are stored into (or between) the computational elements of the platform, and the time spent on doing the actual *computation*.

To design a computer system that provides the best time-to-solution within a given budget (be it of monetary nature or energy) is a balancing act. Where some platform A may be easier to program at the cost of lower performance, some platform B may perform better at the cost of a higher design time. When the time-to-solution and other economical factors for both platforms break even, we could speculate that in most cases, platform A will still be selected, since machine labor is often preferable to human labor.

1.1.3. SINGLE AND MULTICORE PROCESSORS

What platforms and tools are available to developers to minimize the time-to-solution in the context of analysing large volumes of digital information with computers? Answering this question requires a dive into the past.

Today, virtually every computing platform is implemented with complementary metal–oxide–semiconductor (CMOS) technology. (C)MOS technology is the main driver behind the success of computers in general over the past half century. Over several decades following the 1970's, the mainstream computational platform used by most developers were integrated circuits (IC's) created using the (C)MOS fabrication process. These IC's typically held a single-core general-purpose processor, also seen, greatly simplified, in Figure 1.2. One could program it to perform any computational task through a fixed set of simple (and later more complex) instructions. First helped by assemblers and later compilers, the tools and the platform provided the developers with a low time-to-

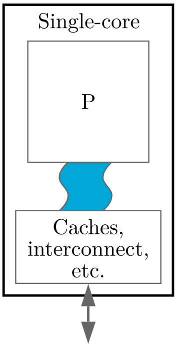


Figure 1.2: A single-core processor. P: Processor core. Implementing applications involves creating the correct, single stream of instructions alongside the data.

solution to computational problems, especially compared to creating custom computers with register-transfer level designs out of discrete digital circuit components. The computer industry matured, and by the grace of their main product, the single-core platform, the Information Age was spawned, producing some of the now-largest companies in the world.

Although a large variety of single-core processors are available, the degree of freedom a developer has when implementing applications on such a platform is relatively low, since they 'only' have to decide what the correct sequence of instructions must be to produce the desired output. The developer does not care much about the details of the architecture of the digital circuit, merely selects a chip that adheres to some more macroscopic requirements, such as the instruction set architecture, potential computational performance, power usage, and cost.

Having relatively little freedom of only being able to define the instruction sequence, compared to designing a chip from the ground up, may seem limiting. It can also be seen as empowering, since no more time has to be spent on the many of the decisions required to design the processor — work typically done by large teams of engineers with the associated risks and investments already taken by the companies developing the chips. The decreased design time, at the potential loss of performance from the ability to customize the architecture of the chip, is in the vast majority of the cases beneficial to the time-to-solution. The single-core processor reigned supreme for many years.

However, around the turn of the century, it became apparent that there were physical limitations to scaling down CMOS technology to make the single core more powerful (specifically, unmanageable levels of power density; the Power Wall). Furthermore, improving single cores by 'throwing more transistors at it' is subject to Pollack's rule [2], stating that the performance improvement one gets is approximately only the square root of the number of transistors added, making it less interesting to improve the single-core processor itself. This caused a move to multicore designs in the decade following, with core counts slowly but steadily increasing. This type of processor is now the mainstream type of integrated circuit in big data systems, and is shown in Figure 1.3.

A lot can (and has) been said about how, from an architectural perspective, multicore

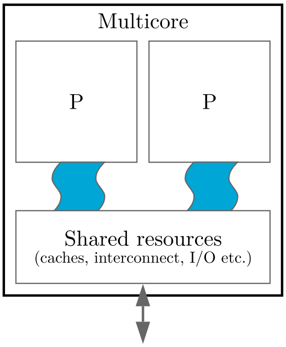


Figure 1.3: A multi-core processor. P: Processor core. Implementing applications involves creating multiple sequences of instructions alongside the data, while making sure no conflicts occur when sharing resources.

processors decrease the run time in the total time to solution. But for humans to capture a description of a solution based on these new platforms (e.g. to write concurrent programs) appears less trivial, and initially it can be argued that whomever (from a pool of mainstream developers) tried to use this platform, experienced a higher design time. It is much more complicated to describe and debug a solution based on multiple (but still identical) components working together that share information and resources among each other, rather than for a single component that doesn't share information with peers and has all available resources for itself. This is illustrated by two instruction and data streams entering the processor cores in Figure 1.3; rather than having to define one sequence of instructions, a developer needs to define multiple that operate concurrently, sharing the same resources.

It took some time following the introduction of multicore processors for highly productive tools to become available; tools that inherently expose and abstract the concurrent capabilities of the platform. These tools often appear in the form of programming languages or extensions thereof. For example, parallel programs can be written relatively easily in the C language using the OpenMP extension. OpenMP abstracts platform-specific details when multiple cores work on the same problem in parallel threads (e.g. how to schedule the threads and how to communicate between threads). OpenMP is a pragmatic solution, and is not inherent to the C language. Using its constructs requires to literally prefix them in code with the `#pragma` directive.

More elegant tools came into existence over time, taking concurrent and parallel programming into consideration from the drawing board. For example, the Scala language knows a strong notion of immutability, data-parallel collections, provides the means to express functional transformations without side-effects, and more features to enable developers to capture parallel programs more efficiently. When transforming a data-parallel collection by mapping a function onto every element, this can automatically be done in parallel, leveraging the multicore processor's computational power with negligible impact on design time. Another relatively young language named Rust, provides explicit ownership semantics and an extensive type system to guarantee safe (i.e. less error-prone) concurrent and parallel programming. Incorrect concurrent or parallel code can simply not be compiled, resulting in useful error messages for the developer to fix problems before they even arise during run time. These modern and highly productive tools have drastically reduced the design time for applications on a multicore platform.

From the shift to multicore systems we may learn that mature, paradigm-fitting tools take some time to age. New technologies arise that attempt to overcome limitations of older paradigms. New platforms are introduced by industry based on this new paradigm, and new tools are developed. This is initially done in a pragmatic way, causing a relatively high design time still, perhaps worth the improved performance. Only after mapping old and new applications to the new platform is extensively explored does it become clear what sort of tools are needed and how humans may interact more efficiently with them.

1.1.4. HETEROGENEOUS COMPUTING

While the need to analyse more data in less time steadily increases, fundamental limits in the backbone of the world of computing — the semiconductor industry — are in sight. Today, CMOS technology is burdened by the slowdown of Moore's law and the failure of Dennard scaling, causing chips to easily approach the limits of the power [3] and transistor budgets. The amount of data humanity gathers, and the computational resources required to process it, keeps increasing. Unless better techniques to produce integrated circuits are found, this provides an ill omen to satisfy the computational needs of big data analytics applications in the future.

However, at the level of the digital circuit architecture, there still seems to be some room to play. For example, it has been theoretically shown that depending on the amount of exploitable parallelism in a workload, given the same budget, a combination of a large, fast processor core, with many slower but less costly processor cores can be more effective than several large processor cores [4]. Digital circuits can furthermore be specialized to more specific tasks rather than be organized for general-purpose computing in its broadest sense. Through specialization, every transistor and unit of energy can be used more effectively towards a more specific computational goal. With this perspective, it is not surprising that an often mentioned successor to the multicore platform — the heterogeneous computing platform — has arisen.

In heterogeneous computing, the computing platform exists not just of multiple identical computational cores, but also of *different types* of cores, adding more performance not only through numbers but also through *specialization*. Commercial examples started with the Cell Broadband Engine, found in the PlayStation 3. Here, apart from a general-purpose processor, also smaller but more specialized computational cores were present that perform well in areas such as e.g. physics simulation and multimedia workloads. While these specialized computational cores may reside on the same CMOS IC as in the Cell, in data centers today, they are commonly found on a different chip and integrated through the use of high-bandwidth interconnections. They typically reside on a printed-circuit board that is connected to the main board of a host processor, connected through a peripheral bus, such as PCIe. This style of component is today often called an *accelerator*.

The most prominent contemporary example of such an accelerator is the graphics processing unit (GPU), also shown in Figure 1.4. Around the end of the previous decade, the instructions that the tiny but incredibly numerous cores of a GPU could perform

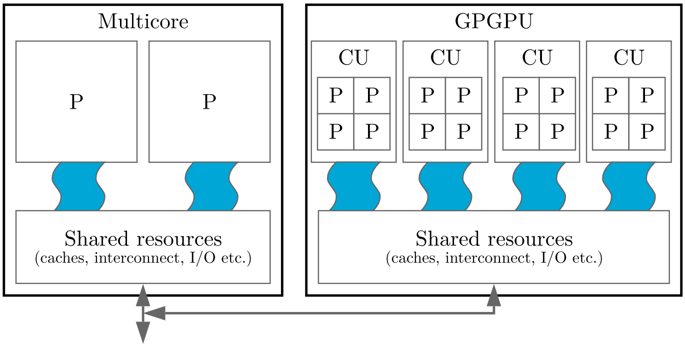


Figure 1.4: A typical graphical processing unit allowing general-purpose computing (GPGPU) connected to a multicore processor. CU: Compute Unit, P: Processor core. Implementing applications involves creating multiple sequences of instructions alongside the data, while making sure no conflicts occur when sharing resources. Processor cores within a compute unit work in parallel with the same stream of instructions, but on different parts of the problem. Different compute units can work on different instruction streams.

were generalized, allowing to perform any sort of computation rather than having many different types of tiny cores that could only perform very specific functions. Therefore, GPUs, were made more efficient to not just perform graphics rendering related tasks, but also to do more general-purpose highly parallel computations. Companies then offered their so called general-purpose GPUs (GPGPUs) to all developers, not just developers that were solving graphics rendering problems.

Heterogeneous systems using accelerators are more complex, as developers must make more decisions. This is illustrated in the case of a GPGPU-accelerated system in Figure 1.4. When the system architecture changes, all components of the time-to-solutions must be reconsidered.

Since data to be operated on traditionally resides in the memory of the host system of some accelerator, it must be moved over a relatively large distance to the accelerator to be operated on. Therefore, while accelerators are designed to decrease the computation time, the time spent on data movement may increase. The exploration of this problem has led to the well-known Roofline model, providing an intuitive means of making trade-offs as to whether it is worthwhile to off-load some part of the program to an accelerator [5].

Luckily, while the performance of CPUs or even GPGPUs does not increase as fast as it used to, due to the limitations of CMOS technology described at the beginning of this section, the performance of interconnect, network and storage technologies *has* increased over recent years (and at the time of writing still is increasing). A traditional assumption that was sometimes made, that CPUs are fast and I/O is slow, often does not hold any more [6]. Applying the lessons learned from the Roofline model, this means we obtain a steeper slope towards saturation of computational throughput for accelerators, not requiring a tremendous amount of computations per byte (arithmetic intensity) to make moving the data worthwhile. This paves the way for more workloads to be off-loaded to accelerators efficiently, since the overhead of data movement is being lowered relative to the computation.

Another aspect of the time-to-solution of the heterogeneous computing system is the design time. At the time of writing, the tools for GPGPU programming are in their adolescence, as they usually still require pragmatic solutions to expose their functionality to the designer, e.g. through pragmatic constructs in languages from an older paradigm (OpenACC), by specializing existing languages (e.g. C++ on the host CPU with CUDA flavored C/C++ kernels for the GPGPU), or through libraries greatly abstracting their use for specific application domains only (e.g. TensorFlow for machine learning). Luckily, there are already some new languages that take GPGPUs into consideration as mainstream components, and have included syntactically and semantically pleasing abstractions and constructs into their design from scratch (e.g. Halide [7]).

1.2. Field-Programmable Gate Arrays

Within the context of heterogeneous computing with accelerators, a component that may be experienced (from a mainstream software developer point of view) as radically different is getting an increased amount of interest: the Field Programmable Gate Array (FPGA), also shown in Figure 1.5.

FPGA devices allow the implementation of an arbitrary digital circuit, by appropriately configuring an immense amount of fine-grained customizable logic blocks, arithmetic units, memories, input/output blocks, and on-chip interconnect resources. As such, a developer can completely specialize the digital circuit to perform exactly (and perhaps only) the desired function. Through specialization, it is possible to achieve decent performance, even though because of the underlying technology, the clock rates of FPGAs are an order of magnitude lower than CPUs and they may use between two to over a hundred times more transistors to implement the same function, depending on the resource [8]. The FPGA may shine in applications that were not (yet) economically viable to include dedicated circuits in CPUs, GPGPUs or application-specific integrated circuits (ASICs). For example, this dissertation will describe several such applications in Chapter 4, where e.g. a new type of floating-point arithmetic is explored, that can on CPUs or GPGPUs —for now— only be emulated through software.

Originally used for rapid prototyping of digital circuits, FPGAs have proven to be useful components in the embedded systems domain, where they are often used as a highly connective and flexible solution for timing-critical or performance-critical applications. With the release of datacenter-oriented accelerator cards by major FPGA companies such as Xilinx and Intel, and the offering of FPGA-enabled instances by cloud providers such as Amazon, Nimbix and Microsoft, however, it seems that the industry is trying to push these components into the data center and cloud, allowing the broader audience to make use of them.

As with the introduction of multicore processors and GPGPUs, we must carefully consider the time-to-solution for the FPGA accelerator platform. Implementing applications on FPGAs is not a matter of using the right pre-defined *instructions* to let the circuits found in one's CPU do its job in such a way that the correct output is produced. Rather, it is a matter of coming up with the most effective *circuit* such that the correct output is produced. The platform is readily available today, but leveraging its capabilities efficiently is still left to a small set of expert developers, typically with a hardware-oriented background. Although its performance can be very good in some cases, contemporary tools to program

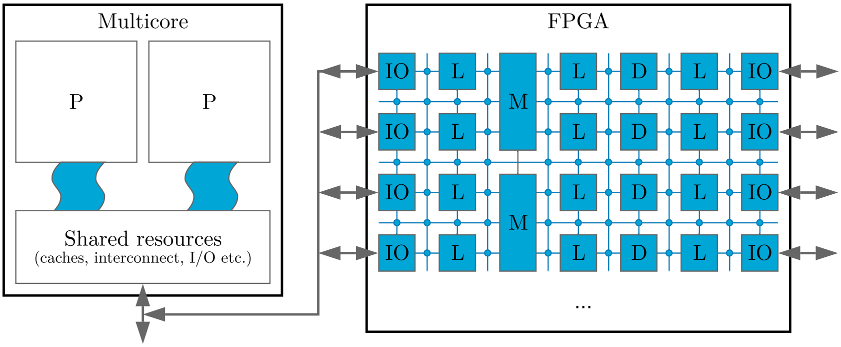


Figure 1.5: A typical field-programmable gate array (FPGA) connected to a multicore processor. IO: input/output blocks handling external signals, L: logic blocks to implement simple boolean functions and registers. M: memory blocks to implement relatively large on-chip memories. D: DSP blocks with complete arithmetic units for integer and floating-point computation. Implementing applications involves configuring and connecting the aforementioned blocks in the right way, allowing any sort of digital circuit to be mimicked, even processors themselves. This gives the developer a great amount of freedom, but also requires a specific set of skills and a great amount of low-level architectural choices to make. This contributes to a high design time component in the time-to-solution.

the platform are still quite hard to use. Thus, even if the platform itself can provide high computational performance, the time-to-solution (for the mainstream developer) is still larger than for other platforms with more matured tooling, simply because the design time is very high. The high design time is tightly related to the radical difference between FPGAs and GPGPUs and CPUs.

To understand why the difference is described as radical, we need to know how the mainstream software developer in the big data domain designs solutions for the components found in the data center. Today, this component is still predominantly a general-purpose processor. Virtually every commercially successful mainstream general-purpose processor works by processing a sequence of instructions as shown in Figures 1.2, 1.3 and even for the GPGPU of Figure 1.4, the same could be said, with the appropriate nuances. The available functions in the hardware of the processor are fixed, but it may be made to do different things by correctly feeding different instructions. Thus, the developer ‘simply’ needs to properly place instructions from the instruction-set in the right order and feed them to the processor, alongside the data to operate on. The instruction-set abstracts the underlying hardware mechanism by which the sequence of instructions are executed.

Determining the bits and bytes that represent the stream of instructions by hand is not productive — the design time is too high. More abstraction is required to decrease the design-time; assemblers abstract the instruction bits to somewhat human-readable assembly languages, where one can define the sequence of instructions more easily. Compilers abstract the assembly language to human-readable programming languages, where we still (although somewhat indirectly) define the sequence(s) of instructions.

But there is a lot of commonality amongst solutions using computers; they all require some input/output, management of memory, a means of starting up and shutting down, interfacing with the human, etcetera. This is provided by another layer of abstraction; the operating system. Now there are many different types of platforms and operating systems that solutions could run on. If they are different, developers have to write the programs for each platform separately, leading to increased design times. Therefore, language run-time engines have been created that mimic processors on processors (e.g. virtual machines, such as the Java Virtual Machine). They map an intermediate instruction set to the platform-specific bare-metal instruction set. Now the time to solution decreases when multiple different platforms exist as an implementation target.

Still, the developer yearns for more abstraction. Language run-times exist that even go as far as to not even compile the source code to a sequence of instructions anymore. One can now simply provide the sequence of strings that is the description of the program, and it only gets interpreted as the program runs (through interpreters, such as CPython). The interpreters call the appropriate pre-compiled instruction sequences such that the desired functionality is eventually materialized many layers down in the hardware. Environments with many dependencies may differ between various systems that a developer will potentially use. So called containers package whole application environments so they may be easily installed and run on any sort of system. The use of such virtual machines, interpreters and containers is commonplace in the field of big data analytics today, with good reasons. This will be discussed in more detail in Section 1.3.

The bottom line is that it seems that before the end of Moore's Law, there was a lot of space to provide sometimes costly abstractions in the field of computer science and engineering (although this space is decreasing). The many innovations have added so many layers of abstraction that developers now use computing platforms without thinking about currents, transistors, bits, gates, bytes, memories, instructions, operating systems, or user-space environments.

Back to the world of FPGAs. These components do not implement a common instruction set that can easily be made human readable through an assembly language. The programming languages used for FPGA designs are inherently concurrent, where a sequence of statements cannot be read like some causal 'story' where first a thing happens followed by the next, sequentially changing the state of the system. The source code must be written, read and interpreted like a circuit diagram, where everything happens at the same time. FPGAs allow to implement arbitrary digital circuits by combining lookup tables (LUTs) for boolean functions, flip-flops to hold state, and extensive interconnection networks for arbitrary connections between the previous. There are no equivalents of mainstream operating systems managing memory, I/O devices, and security. There are no equivalents of mainstream virtual machines, interpreters or containers to abstract the wide variety of platforms and environments.

Therefore, the level of abstraction that a developer typically sees when attempting to design an application with an FPGA is that of the digital circuit. While designing an application at the level of the digital circuit provides a massive amount of freedom, it also requires *a massive amount of choices* to make as well; difficult architectural choices that were made for the developer already when working with mainstream a general-purpose processor and the whole stack of software built on top of it.

To make effective choices in FPGA design, a very specific set of skills is required, related to digital circuit design, computer architecture and system integration. It would be safe to assume that the mainstream software developer found in the big data ecosystem does (understandably) often not possess this skill set. They are trained to work on top of the many layers of abstractions that have been developed over the course of many decades, and for good reason; the layers are many and complicated, and every person, team or even company has limits to the amount of layers of abstractions at which they can develop applications.

There is a large body of work in research and industry trying to coerce descriptions of software programs written in traditional software languages (such as C/C++) into automatically synthesized hardware designs, through a technique often called High-Level Synthesis (HLS). Unfortunately an extensive amount of vendor-specific hardware-oriented pragmatisms are still required to be added to the code to produce functional and, with considerable effort, performant designs, hampering both the time to design and the time to run solutions. HLS seems interesting for those developers trained in both hardware and software development and want to rapidly create functioning hardware implementations, not caring too much about performance. It is, however, still unlikely that the mainstream software developer (nowadays not caring about bits, bytes, or even managing memory anymore) would be drawn to this approach. It arguably seems that the rate at which the level of abstractions in mainstream software languages rises is much higher than the rate at which HLS tools improve, creating an increasing gap in productivity. Furthermore, the arguably awkward abstractions from software-oriented languages applied to digital circuit design often cause a loss of performance compared to hand-coded hardware designs [9]. In an ecosystem where the direct competitors are highly optimized CPUs and GPGPUs, not burdened by low clock rates and circuit overhead like FPGAs, the loss of performance versus decreased development time is a trade-off again concerning the total time-to-solution.

In general we therefore argue that in the near term, it is more likely that as FPGA accelerators become widely available in cloud infrastructures, experienced hardware developers will provide well-engineered high-performance solutions for common problems in data centers and big data analytics, with useful application programming interfaces (APIs). When the tools to engineer high-performance FPGA solutions become more productive, the widespread acceptance and use of FPGA accelerators in big data systems will be accelerated, as more efficient solutions can be produced in a lower amount of time by the experts. Not only ’hyperscalers’ with enough resources to hire large teams of hardware engineers would be able to rapidly produce high performance solutions on heterogeneous systems with FPGA accelerators, but also smaller companies that are tenants of cloud-based FPGA infrastructures could consider hiring smaller teams to improve performance-critical parts of their pipeline.

Because of the vast amount of choices available when working with the FPGA accelerator platform, abstracting the choices to more productive constructs is more easily done in a domain-specific context. We will demonstrate an example of a tool providing such abstractions in the context of big data analytics systems working on tabular data structures in this dissertation, especially in Chapter 3.

To embrace FPGA accelerators in big data analytics, interfaces must be provided in the languages used and loved by the existing community, with APIs that match the level of abstraction of the existing ecosystem in which they should be integrated, something we heavily focus on in this dissertation in general. The recently released Xilinx' Vitis framework also moves in that direction quite well, by providing interfaces for languages heavily used in the big data ecosystem, such as Python.

Also, for data-intensive workloads, data should be able to move over these interfaces at high bandwidth to match the increasing I/O bandwidth of contemporary and upcoming accelerator systems, otherwise the accelerators will not be able to live up to often advertised performance metrics derived only from their computational performance, due to the implications of the Roofline model. The ability of such interfaces to achieve system bandwidth in the order of tens of gigabytes per second is a central theme in this work.

To summarize, the time-to-solution is still rather high for FPGA accelerated platforms. The community should strive to:

- Provide better hardware design tools with proper hardware-oriented abstractions that do not hamper the means to obtain the intended performance by leveraging FPGA technology as well as possible.
- Match the level of abstraction of the interfaces to the accelerated solution with those of the ecosystem they are to be integrated in.
- Provide integral solutions that take into account all aspects of the platform architecture and time-to-solution.

1.3. BIG DATA ANALYTICS

To progress towards explaining the relation between FPGA accelerators and big data analytics in more detail, we must also consider the historical perspective of the field of big data analytics. Besides considering the computational components of the several hardware layers of big data systems, we must also consider the numerous layers with the software components of these systems.

Over the course of the last two decades, it became increasingly evident that data sets grew so large that no single traditional computing system (e.g. a single data center node) could be reasonably equipped with enough resources to store and process the data set. Therefore, it was required to create a network of multiple nodes allowing to *scale out* the computational platform.

Scaling out was already applied to supercomputers in the domain of High-Performance Computing (HPC). Then why is big data not considered the same as HPC? The distinction between HPC and big data has been (and probably still is) a hot topic of debate. The discussion could be summarized by saying HPC systems are more centered around achieving as high as possible computational throughput for mainly scientifically-oriented simulations, while big data systems are more centred around quickly extracting value from massive amounts of existing data. HPC technologies were typically designed to work in specially engineered on-premise compute clusters, that have a team of dedicated engineers managing the clusters and the software running on them. HPC-oriented software typically leans heavily on the MPI library for low-level languages such as C, C++ and Fortran. On the other hand, big data systems are *much* easier to use for developers, typically designed to work on commodity hardware, and integrate well with existing database systems and cloud infrastructures without the need to drastically redesign the

```
1 val textFile = sc.textFile("hdfs://...")
2 val counts = textFile.flatMap(line => line.split(" "))
3                  .map(word => (word, 1))
4                  .reduceByKey(_ + _)
5 counts.saveAsTextFile("hdfs://...")
```

Figure 1.6: Counting words with Apache Spark and Scala [https://spark.apache.org/]

implementation when switching between various clusters. Most applications are written in high-level languages with highly automated run-time engines such as Java, Scala or Python.

To demonstrate how easy it is to program on top of a framework for cluster computation geared towards big data analytics, consider the canonical code snippet shown in Figure 1.6, taken from the website of a well-known cluster computing framework called Apache Spark. This snippet implements a program that counts the occurrences of every word in a text. In this figure, we observe a mere 5 lines of code, allowing the developer to express that they want to load a file (line 1), split every line of the file into separate words (2), construct a tuple of the word with the *value* 1 (3) and consider the word as a unique *key* to aggregate all values for each key by summing them (4), finally saving the file to persistent storage (5).

Not even taking into considering the layers of abstraction discussed in the previous section, behind the scenes of these five lines of code running on a cluster, a large amount of domain-specific features are in play. Under many layers of abstraction, these features are 'hidden' from (i.e. not explicitly exposed to) the developer, allowing them to relatively easily develop big data analytics applications that scale well. In this example alone, behind the scenes, we may observe the following:

- The text file may be in the order of many terabytes in size, distributed over the storage resources of thousands of nodes.
- The construction of a lazily evaluated Directed Acyclic Graph describing the steps of computation and dependencies, that is optimized, planned, scheduled, distributed, and executed in parallel on the cluster automatically.
- In the reduction step, the tuples are automatically shuffled across multiple nodes in the cluster, such that the same keys end up in the same place to be able to perform the reduction.
- The implementation is resilient to node failure, such that when nodes fail, their work is redone elsewhere, providing a certain level of fault-tolerance.

The many layers of abstraction provided by the big data analytics frameworks such as Spark are made possible through highly productive programming languages and their underlying infrastructure

Continuing with the example of Apache Spark, its main goal is to provide a scalable environment on top of clusters of commodity hardware, customized hardware, cloud infrastructures or a mix of the previous. It is therefore unsurprising that, initially (and perhaps still), it seemed a good idea to program Spark in a language running on virtual machines, such as the Java Virtual Machine (JVM), in order to make the framework platform-agnostic. Designed long before the dawn of the big data era, the technological choices in the JVM were not driven by data-intensive workloads, but by the ability to be platform-agnostic, and allow for more productive programming languages.

However, it has become evident that some of the default techniques used by virtual machines or interpreters (such as CPython) do not favor the run time component of data-intensive workloads. Garbage Collection (GC) to provide automated memory management is one example. The JVM automatically manages objects that are dynamically created during the execution of a program. While they are explicitly created by the programmer, they do not have to be explicitly deleted from the memory by the programmer anymore, such as in C++. The GC mechanism will keep track of all unreferenced objects, and will automatically delete them periodically or when running out of memory. It will also move objects to prevent fragmentation of the memory assigned to the JVM. However, for large data sets with sizes in the order of the available system memory, moving the data around frequently seems rather wasteful of time and energy.

GC technology spawned a massive amount of research trying to decrease its cost by obtaining deeper understanding of program behavior, and applying various flavors of the GC technique corresponding to the specific behavior. Despite these valuable contributions, the limits of GC seem so fundamental in data-intensive applications, including those often programmed on top of Spark, that programmers have ironically decided to circumvent the automated memory management system by storing the bulk of the data outside the memory heap managed by the JVM.

This is not an argument against the use of the JVM in general, rather an argument to not forget the lessons learned from Figure 1.1. Since when we look back at the components of the time-to-solution in that figure, the languages that are used on top of it provide benefits that cause the decrease in design time to outweigh the increase in run time. Developers don't (theoretically) have to re-capture and debug their code when switching to different hardware or operating systems that may be found e.g. over various cloud infrastructure offerings. A relevant quote is from one of the original developers of Spark, Mattei Zaharia: "Even though I like performance [...] ease of use matters more. [...] The biggest performance improvement is when you go from not working to working" [10]. With the Spark project nearing a million lines of (non blank, non commented) code, its sheer complexity also favors this choice. It would be a challenge to just imagine the amount of code that had to be written, verified, and maintained, were this project implemented in a language typically more performant but less productive towards achieving functional programs as quickly as possible, like C.

At the same time, without going into too much detail yet, more technological tradeoffs in high-level language run-time engines were made in a time where big data was not a dominant use case for these engines. We stipulate two more aspects that are of specific interest to this dissertation, that of in-memory layout of the data sets and the presence of language/run-time specific metadata. Modern high-level languages typically also automate the design of the in-memory layout of objects, which may not correspond well to how data-intensive workloads may make more efficient use of the underlying hardware. Also, language/run-time specific metadata is present that may not be of interest to components designed in other languages, or even other sorts of computational components. Thus, this metadata must be removed and the data must often restructured into a more usable format when passing data sets between heterogeneous processes, such as e.g. between Python and Java, or between Java and some hardware accelerator. As we will demonstrate in this dissertation in Chapter 2, the time to restructure the data before being able to communicate it, a process called *serialization*, can furthermore cause serious performance bottlenecks in the path from the software process to accelerator.

1.4. Problem description and scope

We summarize the discussion of the previous sections with the following points, with the latter two providing a problem description that this dissertation aims to explore.

- To keep providing society with answers sourced from the vast amounts of digital information, there is a need for more computational performance to be able to process larger volumes of data with a reasonable time-to-solution.
- Fundamental limits of CMOS technology have caused a slowdown in the performance increase of general-purpose processors. Heterogeneous components, such as GPGPUs and FPGA accelerators provide alternatives for more computational performance through architectural specializations geared towards a specific domain of problems. The throughput of interconnections, storage and network is increasing, thereby also increasing the value accelerators can provide.
- FPGAs have recently become publicly available in offerings of cloud infrastructures and data centers, where a significant portion of the desired computational work will be performed in the foreseeable future.
- **Problem 1**: For the intended use case of FPGA accelerator systems in existing big data analytics systems, there is a high mismatch in the level of abstraction at which both systems are programmed and operated. Data must pass through numerous layers of abstractions that may be detrimental to the performance.
- **Problem 2**: Developing FPGA-accelerated implementations of big data applications has a high time-to-solution, because a developer must make many low-level architectural decisions, and there is little standardization at a high level of abstraction.
- **Problem 3**: FPGA tools are highly vendor-specific, hampering the growth of an open-source community around the technology, which is favored by the existing big data ecosystem.

Since this dissertation encompasses a wide amount of topics and technologies, we continue to scope the topics of interest as shown in Figure 1.7. While all topics are of immediate interest to the general theme of this dissertation, we find it useful to explicitly

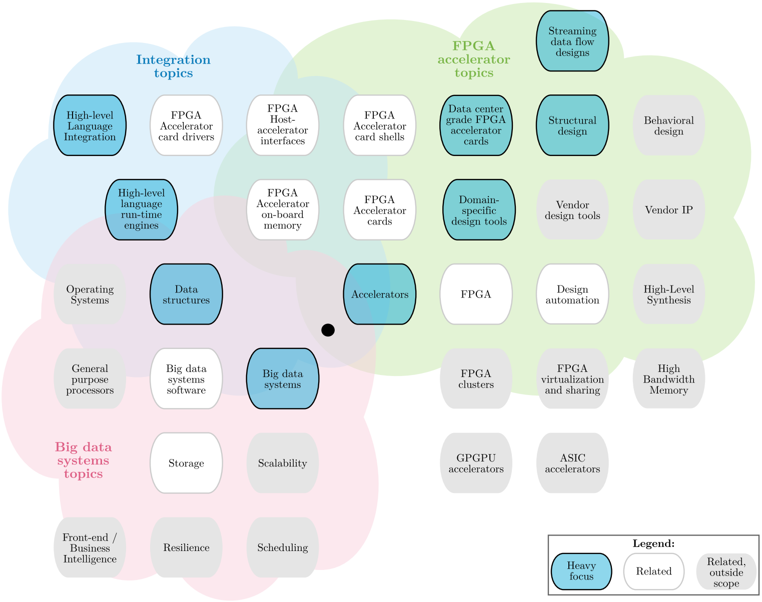


Figure 1.7: Related topics and scope of this dissertation.

declare what topics enjoy a heavy focus and are explicitly contributed to. We also declare what topics are related and somehow impact this work, but that we do not explicitly contribute to. Finally, there are topics of general interest, but they are not of immediate impact to this work, and are typically briefly mentioned and discussed only.

We focus on FPGA accelerators and their integration in big data systems software. We focus on how data structures in the context of big data system are currently represented in the software components and their high-level language run-time engines. We focus on how transporting data structures between FPGA accelerators and the run-time engines can be made efficient. We focus on how the hardware structure to efficiently transport such data structures into FPGA accelerator kernels may be automatically generated through domain-specific design tools, but we do not focus on behavioral kernel implementation. We focus on open-source and freely available tools that support a hardware-description language design flow, but do not focus on vendor-specific tools or components that are vendor-specific IP.

To integrate FPGA accelerators with big data systems software, we must take into consideration FPGA accelerator cards, their drivers, their top-level shell designs, the host-to-accelerator interface systems through which they are typically connected. GPGPUs and ASIC accelerators are outside the scope of this dissertation, although we present hardware design methodologies that may be applied in ASIC design as well.

We do not focus on any front-ends, business intelligence, or end-user applications. We do not focus on how FPGA devices may be clustered for big data applications, e.g. such as in [11]. We do not focus on how FPGAs may be virtualized and shared among multiple tenants of the cloud infrastructure. We do not focus on scheduling for FPGA-accelerated applications on top of big data cluster computing frameworks. We briefly deal with storage, but only in the context of an accelerator to decode a storage format.

1.5. CONTRIBUTIONS AND OUTLINE

Related to the problems and scope described in the previous sections, the main research question that this dissertation aims to shed light on is as follows:

How can FPGA accelerators be efficiently integrated with contemporary big data systems software?

Because the word *efficiently* may be considered ambiguous, we explicitly mention that it relates to the time-to-solution as a whole as described in Section 1.1.2. With this question in mind, the outline of the remainder of this dissertation is as follows, where we pose several related questions:

- **Chapter 2**: To answer the main question we first analyze the underlying technologies of big data systems software to expand on **Problem 1**. In Chapter 2, we explore the question: **What challenges arise from the desired merger of big data systems software and FPGA accelerators?** We explain that many of the software systems depend on virtual machines and interpreters, and discuss in detail some of the techniques that impact the time-to-solution, especially on the side of the run time. A specific challenge of serialization overhead was tackled by the community alongside the work of this thesis. The project that addresses this problem, named Apache Arrow, is widely used in this dissertation. We also discuss the relevant details of that project.

- **Chapter 3**: We furthermore deal with the design time aspect of the time-to-solution as described in **Problem 2**. Because of the vast amount of choices that have to be made when designing FPGA accelerated systems, we find it useful to ask the question: **What of an FPGA accelerator design and software interface can be automated in the context of big data systems?** This has led to the development of an extensive toolchain called Fletcher, helping to automate FPGA accelerator design an integration in the domain of big data applications working on tabular data sets, presented in Chapter 3. To address the large mismatch in level of freedom and available open source tooling in the big data analytics ecosystem and FPGA accelerator design as mentioned in **Problem 3**, we furthermore ask the question: **(How) can a platform-agnostic environment be created in the currently highly vendor-specific context of FPGA accelerator design?**

- **Chapter 4**: Various big data analytics applications were implemented to exploring and expanding the implementation of the Fletcher toolchain, to answer the question: **What applications can benefit from the features of the Fletcher framework?** These applications are presented in Chapter 4. Some applications furthermore demonstrate the particular merit of FPGA accelerators in general, achieving higher performance through specialization than contemporary CPU systems. We have developed applications using the contributions of Fletcher in various sub-fields of big data analytics, including text analytics (regular expression matching), genomics (variant calling), machine learning (clustering), and storage (file decoding).

- **Chapter 5**: Based on the contributions in the Fletcher toolchain, we have continued to explore **Problem 2**. More specifically, we explored the possibility to decrease the design time for hardware designs working on complex data structures, as are commonly found in big data analytics. This has raised the question: **Can we decrease the complexity of describing interfaces between hardware components that exchange complex data structures?** The final contribution presented in this thesis in Chapter 5 is an answer to this question in the form of an interface specification for streaming dataflow designs transporting complex and dynamically sized data structures.

- **Chapter 6**: In the final chapter, we summarize the answers to the specific questions raised by the previous points, stipulate various directions for future research, and conclude this dissertation.

REFERENCES

[1] M. Chen, S. Mao, and Y. Liu, *Big data: A survey,* Mobile Networks and Applications **19**, 171 (2014).

[2] S. Borkar, *Thousand core chips: A technology perspective,* in *Proceedings of the 44th Annual Design Automation Conference*, DAC '07 (ACM, New York, NY, USA, 2007) pp. 746–749.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, *Dark silicon and the end of multicore scaling,* in *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (2011) pp. 365–376.

[4] M. D. Hill and M. R. Marty, *Amdahl's law in the multicore era,* Computer **41**, 33 (2008).

[5] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures,* Commun. ACM **52**, 65–76 (2009).

[6] F. Kruger, *Cpu bandwidth – the worrisome 2020 trend,* (2016).

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, *Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,* in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13 (Association for Computing Machinery, New York, NY, USA, 2013) p. 519–530.

[8] H. Wong, V. Betz, and J. Rose, *Comparing fpga vs. custom cmos and the impact on processor microarchitecture,* in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11 (Association for Computing Machinery, New York, NY, USA, 2011) p. 5–14.

[9] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, *Are we there yet? a study on the state of high-level synthesis,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**, 898 (2019).

[10] M. Zaharia, *The future of big data (talk),* (2016), 40 Years of Patterson Symposium.

[11] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, *A cloud-scale acceleration architecture,* in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016) pp. 1–13.

2

ANALYSIS OF BIG DATA SYSTEMS SOFTWARE

For accelerators such as GPGPUs or FPGAs to be integrated with big data systems software, it is necessary to study the extensive open source ecosystem on which many big data analytics applications are built. In this chapter, we first give an overview of the general challenges related to FPGA accelerator integration. We then analyze the software technologies used to implement the many components of the ecosystem. We find that the large majority of the software components are written in programming languages that run on virtual machines and interpreters. While such systems support highly productive software languages and provide transparent cross-platform portability for applications, they often do so at the cost of efficiency and memory space. In the context of integrating FPGA accelerators, we observe that significant bottlenecks arise when preparing large amounts of data to be exchanged between such software processes and FPGA accelerators. This observation was also made in the context of exchanging data between software processes of different languages, and from attempts to prevent overhead from non-functional data movement during data serialization, the Apache Arrow project was initiated by the community. Arrow provides a common in-memory format such that serialization may be prevented when passing data between heterogeneous processes. We study the format for applicability in the case of FPGA accelerators, and find it highly suitable to be able to saturate contemporary and future accelerator interfaces.

2.1. OVERVIEW OF FPGA INTEGRATION CHALLENGES

Big data system are reaching maturity in terms of squeezing out the last bits of performance of CPUs or even GPUs. The next near-term and widely available alternative for higher performance in the data center and cloud may be the FPGA accelerator.

Coming from an embedded systems and prototyping-oriented market, FPGA vendors have broadened their focus towards the data center by releasing accelerator cards with similar form factors and interfaces as GPGPUs. Various commercial parties offer cloud infrastructure nodes with FPGA accelerator cards attached. FPGA accelerators have also been successfully deployed at a large scale in commercial clusters of large companies (e.g. [1]).

Whether the FPGA accelerator will become as common an implementation platform as GPGPUs in the data center is still an open question. The answer will depend on the economic advantages that these systems will offer; will they provide a lower cost per query? Will they provide more performance per dollar?

In an attempt to answer these questions, valid reasons to be sceptical about embracing FPGA accelerators in the data center exist. We stipulate three disadvantages within this context:

1. Technological disadvantage: FPGAs run at relatively low clock frequencies and require more silicon to implement the same operation compared to a CPU or GPGPU, requiring the specialized circuits they implement to be orders of magnitude more efficient at whatever computation they perform before they provide an economically viable alternative.

2. Hard to program: A notorious property of FPGAs is that they are hard to program, incurring high non-recurring engineering costs; a higher cost per query or more dollars to achieve decent performance.

3. Vendor-specific: Relative to the software ecosystem in the field of big data analytics, one could observe a lack of reusable, vendor-agnostic, open-source tooling and standardization. The big data analytics community has shown to thrive and rely specifically on open-source frameworks, as this provides more control over their systems and prevents vendor lock-in.

On the other hand, valid reasons to be optimistic exist as well, because of the following advantages.

1. Specialization: FPGAs are able to implement specialized data flow architectures that, contrary to load-store architecture-based machines, do not always require intermediate results of fine-grained computations to spill to memory, but rather pass them to the next computational stage immediately. This often leads to either increased performance or to increased energy efficiency, both of which may provide an economic advantage.

2. Hardware integration: FGPAs have excellent I/O capabilities that help to integrate them in places the GPGPU can not (yet) go, e.g. between the host CPU and network and storage resources. This can help to build solutions with very low latency compared to CPUs and GPGPUs.

2.1.1. Hardware design challenges

The two mentioned advantages have the potential to mitigate the first disadvantage in specific cases, which leads us to mainly worry about the problem of productivity. One branch of approaches that the research and industrial community takes to increase productivity is to say: hardware is hard to design while software is easy to program, therefore we should be able to write software resulting in a hardware design. While the term has become somewhat ambiguous, this approach is called High-Level Synthesis (HLS), which we interpret here as; using a description of a software program to generate a hardware circuit performing the same function, hopefully with better performance. A thorough overview of HLS tools can be found in [2].

The HLS approach can (arguably) lead to disappointment on the side of the developer, since it is easy to enter a state of cognitive dissonance during programming. A user with a software design background may find many constructs and libraries not applicable or synthesizable in a language that s(he) thought to understand. Hardware-specific knowledge must be acquired, and often non-portable pragmatism must be applied to end up with a working implementation. A user with a hardware design background may experience a lack of control that may result in sub-optimal designs, hampering the intended performance that they know could be achieved using a HDL. Software languages are designed with the intent to abstract CPU instructions, memories and I/O, but not the gates, connections, and registers that hardware-oriented users desire to express more explicitly than what is allowed by most software-oriented languages. A recent meta-analysis of academic literature in [3] shows that designs created with HLS techniques at a reduced design effort of about 3× still show only half the performance compared to HDL designs, although the meta-study includes designs in frameworks that would classify as an HDL approach (e.g. Chisel) more than HLS, according to our definition. Since the direct competitor is the server-grade CPU and the GPGPU, it is in many cases unlikely that losing half the performance is acceptable.

For the reasons mentioned above, we argue (together with [4][5]) for a different approach to attack the "hard-to-program" problem; hardware is hard to design, therefore we need to provide hardware developers with abstractions that make it easier to design hardware. Such abstractions are easier to provide when the context of the problem is narrow, leading to domain-specific approaches. We must increasingly take care that these abstractions incur zero overhead, since technologically, we are getting close to an era where the added cost of abstractions cannot be mitigated by more transistors, due to the slowdown of Moore's law.

We stipulate three FPGA-specific challenges from the hardware development point of view when designing FPGA-based hardware accelerators for big data systems, that cause a substantial amount of development effort.

H1. **Portability**: Highly vendor-specific styles of designing hardware accelerators prevents widespread reuse of existing solutions, often leading hardware developers to 'roll their own' implementations. It also makes it hard to switch implementations to different FPGA accelerator platforms of different vendors.

H2. **Interface design**: developers spend a lot of time on designing interfaces appropriate for their data structure, since they are typically provided with just a byte-addressable memory interface. This involves the tedious work of designing appropriate state machines to perform all the pointer arithmetic and handle all bus requests.

H3. **Infrastructure**: hardware developers spend a lot of time on the infrastructure or sometimes colloquially called ‘plumbing’ around their kernels, including buffers, arbiters, etc., while their focus should be the kernel itself.

2.1.2. Big data system integration

Not only FPGA-based designs themselves can be very complex — the big data analytics frameworks in which they need to be integrated are very complex as well. For the sake of the discussion in this dissertation, we are going to assume that there is a hardware developer wanting to alleviate some bottlenecks in a big data analytics pipeline implemented in software through the use of an FPGA accelerator. In such a context, it is safe to assume that there is a lot of data to be analyzed. The FPGA accelerator must have access to this data. Where is the data?

Assuming the analytics pipeline to be implemented in the C programming language, a programmer may point to their efficiently packed, hand-crafted `structs`, `unions`, arrays, pointers to nested dynamically-sized data structures, and eventually the primitive types of data that makes up the data structure of interest. Were this data structure to be somewhat inefficiently laid out in memory in terms of feeding it to the accelerator, the programmer would be able to easily modify the exact byte-level layout of the data structure in memory, typically causing the data to reside in regions of memory that are contiguous as possible, such that they can be loaded into the FPGA using large bursts, preventing interface latency from becoming a bottleneck when many pointers need to be traversed. These assumptions are reasonable and describe a common design pattern in hardware acceleration of software written in low-level languages such as C. However, we will show that for the domain of big data analytics, these assumptions usually do not hold.

Where is the data?

We have analyzed the code bases of many active and widely-used open-source projects related to big data analytics. The goal is to answer the question: what languages are mostly used in the big data ecosystem and how do they manage data in memory? While there are hundreds of candidates in the open-source space alone, we have selected projects that are commonly found in the middle-ware of the infrastructure. This is where accelerators are most likely to be integrated. We therefore do not include frameworks focused on specific applications or end-users (e.g. deep learning or business intelligence), since they are often built on top of the middle-ware frameworks that we analyzed.

The overview of the analyzed frameworks is as follows:

- 8 query engines: PrestoDB, Cloudera Hue, Dremio, and Hive, Drill, Impala, Kylin, Phoenix
- 7 stream processing engines: Heron, Samza, Beam, Storm, Kafka, Druid, Flink
- 15 (in-memory) data stores engines: MongoDB, CouchDB, Cassandra, CockRoachDB, CouchDB, OpenTSDB, Accumulo, Riak, HBase, Kudu, Redis, Memcached, Hadoop-HDFS, Sqoop, Arrow

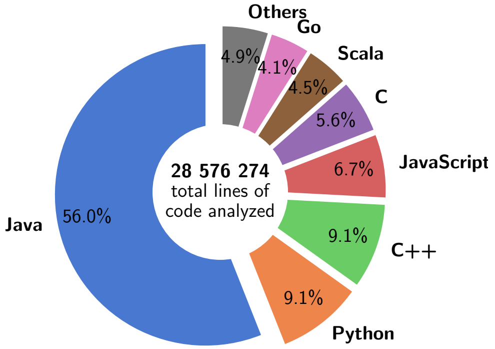


Figure 2.1: Language analysis of 52 open-source projects from the big data ecosystem.

- 9 management and security frameworks: Airflow, ZooKeeper, Helix, Atlas, Prometheus, Knox, Metron, Ranger
- 6 hybrid general-purpose frameworks: Mesos, Hadoop, Tez, CDAP, Spark, Dask
- 4 logging frameworks: Flume, Fluent Bit, Fluentd, Logstash
- 2 search frameworks: ElasticSearch, Lucene-Solr
- 3 messaging / RPC frameworks: RocketMQ, Akka, Thrift

A pie chart of the analysis is shown in Figure 2.1. From the figure, we may find that the vast majority of the code-base is written in Java, followed by Python, with C/C++ taking up about 15% of the lines of code. This gives an indication of the run-time technology used in big data analytics pipelines.

About 80% of the code found in the ecosystem is written in languages that typically alleviate the burden of low-level memory management by various methods that cause several problems. First, garbage collection (GC) is often applied to prevent memory leaks, sometimes causing data to move around the memory, invalidating any pointers to the data, causing the need to halt the software run-time when FPGA accelerators would be operating on the data. Second, extensive standard libraries with containers for many typical data structures (e.g. strings, dynamically sized arrays, hash maps) are commonly used. This decreases the development effort and provides a form of standardization within a language. However, the language-specific in-memory formats of these containers often don't correspond well to how it would be preferable for FPGA accelerators to access the data. Finally, data is often wrapped into objects (e.g. in Python), even though the native architecture supports a specific data type in hardware. What in C is a simple array of a thousand integers would in Python look like an array with a thousand pointers to integer objects, which is not very efficient to access with high throughput, as it is potentially highly fragmented. Furthermore, these objects contain language and run-time specific metadata that are of absolutely no use to an accelerator, such as, for example, pointers to the class of the object in Java.

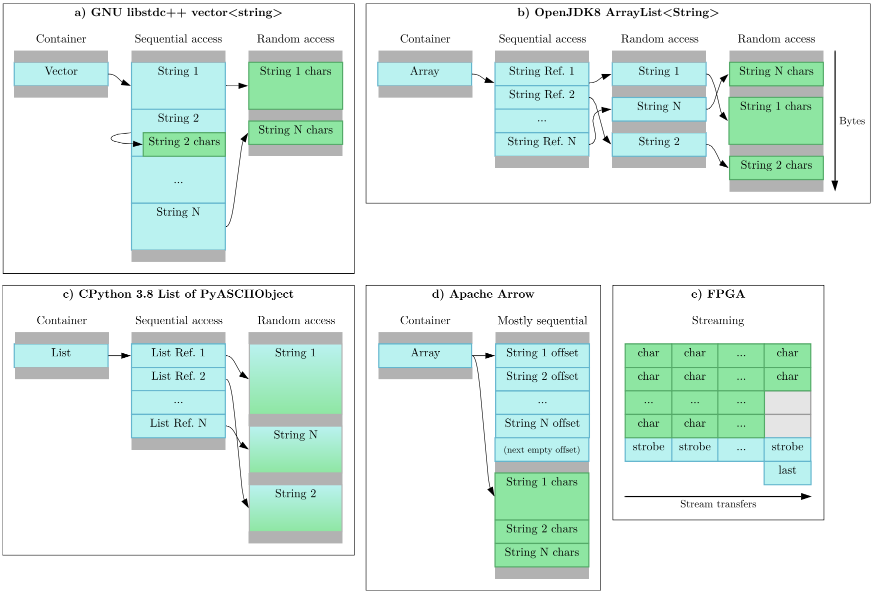


Figure 2.2: Examples of in-memory formats of a collection of strings (a, b, c, d), and how a string could be exchanged between components of a digital circuit in FPGA technology (e). The memory access style when traversing through the collection is described above each layer of reference. Blue regions hold meta data about the data structures, green regions hold the actual contents of the information (the characters).

MAKING THE DATA USABLE FOR ACCELERATORS

When processing a data set with an external accelerator, the data must be moved from host memory to accelerator over its interface. The bandwidth of this data transfer is maximized when the data resides in a large contiguous memory buffer (CMB) because it may be transferred using large contiguous bursts. Thus, a developer who wants to use an FPGA accelerator to speed up some application must first make sure the data resides in a CMB, lest many short transfers with the associated overhead must be initiated. However, most commonly used containers and objects in various languages do not store the data in a CMB. The in-memory formats for such containers and objects are often designed for efficient use within the language run-time itself, or to provide some sort of abstraction that suits the language paradigm well. To prevent accelerators from having to traverse objects graphs, possibly incurring memory latency several times *per object*, serialization must be applied. However, serialization negatively impacts the effective bandwidth to the accelerator.

Consider a seemingly simple collection of data objects of a string type. In Figure 2.2, we show for specific software languages, run-time engines, and standard library versions, what the in-memory format looks like. We will discuss some problems related to the in-memory layout of strings in software systems. We focus on the example of a C++ Standard Template Library (STL) string, since the C++ language generally allows for the greatest flexibility to mitigate the problems, compared to the other languages. While it is possible to allocate the string in an STL vector such that the string objects themselves reside in a CMB, the string object constructor allocates memory for its character array using `malloc()` separately for each string. Thus, the characters (data of interest) of the string *are not guaranteed to reside in a CMB*. This is a general problem in case objects hold variable length data that is allocated by the object itself.

To continue with the example of a C++ STL string object, STL constructors can (in contemporary versions of the C++ language) be provided with custom *allocators* that could (albeit in an arguably counter-productive manner) place them in a CMB. However, if the strings are sufficiently short, the characters are actually placed in the string root object space itself (by both the LLVM and GCC implementation of the STL). This is defined in the behaviour of the constructor, and is an optimization that prevents a second memory allocation from taking place. This effectively breaks up any CMB of characters. It is therefore not possible to guarantee that the data is stored contiguously using an STL string, as we can only dictate that, *if* it allocates, it should use our custom allocator. Without rewriting the string implementation, we cannot change *when* it allocates. Thus, a developer must create some custom representation and implementation of a string, requiring extra effort.

A similar case can be made for even more abstract languages like Python or Java, where this problem is generally worse and less trivial to mitigate for the programmer, as no direct control exists over object layouts in memory. Even if this effort is spent, a data set built up like this will still suffer from more drawbacks.

Even when objects with equivalent fields are stored in a CMB, their in-memory representations are not equivalent among different language run-times, especially due to the presence of run-time specific metadata (e.g. JVM: class references, C++: virtual function tables, Python: reference counters), as can be seen in Figure 2.2. This (to an accelerator useless) metadata may be of significant size, especially when objects are small and numerous, as commonly seen in big data analytics. Therefore, even if the data may be stored in a CMB, *effective* bandwidth is decreased. Furthermore, it is required for an accelerator to implement a filtering step before processing, to make it a true CMB, i.e. not a CMB that also contains language-specific meta-data. This filter step furthermore would depend on the host-side language run-time used, while the function of the accelerator is essentially not different.

Even worse, object layouts are not guaranteed to be consistent inside a language itself. For example, both the Java Virtual Machine [6] and C++ [7] do not specify or restrict how an object is laid out in memory—it is left to the implementation of the JVM and the compiler, respectively. Especially in the case of strings, some run-time engines even implement more advanced tricks to save memory space for often recurring strings, in a technique called *string interning*. Also, compilers may choose to optimize the lay-out, e.g. to improve alignment w.r.t. cache lines in different ways.

Thus, to effectively integrate an accelerator hardware design targeting a heterogeneous environment, the design must:

- be adjusted for every host-side run-time language,
- be adjusted for every compiler implementation,
- put a restriction on the application compiler/run-time, and filter language-specific metadata,

- invent a custom in-memory format for every non-primitive data type, in every language involved, or
- apply the costly act of serialization.

If one standardizes an as-contiguous-as-possible in-memory format and provides interfaces to produce/consume this data in various languages, all these options become unnecessary or irrelevant. As we will discuss in Section 2.3, the Apache Arrow project provides such a solution.

Summary

At a higher level, we summarize the discussion of this section to the following challenges for developers wanting to integrate an FPGA accelerator solution into a software-oriented big data analytics pipeline:

S1. **Complex run-time systems**: is is hard to get to the data, because it is hidden under many layers of automated memory management.

S2. **Hardware-unfriendly layout**: the data is laid out in a way that is most practical for the language run-time system, with a lot of additional bytes containing data that is uninteresting to the FPGA accelerator. A more FPGA-friendly in-memory format of the data structure must be designed, in order to make it accessible to the FPGA accelerator.

S3. **(De)serialization**: Even if one would hand-craft such a format, one would have to *serialize* the input data for the accelerator into that format, and then *deserialize* the result back into a format that the language run-time would understand. The throughput of (de)serialization is relatively low compared to modern accelerator interfaces, and can easily lead to performance bottlenecks [8].

While a thorough discussion of automated memory management techniques is outside the scope of this dissertation, we may observe that in some big data systems software, such as Apache Spark, challenge S1 is dealt with by storing bulk data outside of the automatically managed memory heaps of the language run-time system. While challenge S2 is related, in the next section, we will first focus and quantify the impact of the serialization overhead mentioned in challenge S3 specifically in the context of the interface between host memory and FPGA accelerators.

2.2. Can the JVM Saturate Our Hardware?

With the advance of the big data era, many different big data processing and storage frameworks have been developed. Many of these frameworks are written in languages that use a Java Virtual Machine (JVM) [6] as the underlying platform to execute compiled programs. This allows a cluster to easily scale out, adding nodes of any type of hardware, as long as they can run a JVM. A well known example is Apache Spark [9] which is written in Scala and is generally run on the OpenJDK HotSpot virtual machine.

Although the performance of programs run on the JVM can (in very specific situations) come close to the performance of native implementations, the added layers of abstraction still impose limits [10]. Speeding up JVM applications beyond Just-In-Time (JIT) compilation can be done using native libraries to squeeze out the last bits of performance that the underlying platform has to offer, sacrificing some of the portability of the application. While still a long way to go, the big data field is slowly catching up with the performance known from the high-performance computing (HPC) domain [11]. However, as the end of multicore scaling approaches, scaling up even native CPU performance will be troublesome in the near future [12].

Thus, as big data problems become bigger, there is a need to go even beyond the performance that traditional multicore systems can offer. For this reason, the research and industrial community is looking at other paradigms, such as combining accelerators or near-memory computing with big data platforms. In this section, the focus is on accelerators.

GPGPU computing is currently the most popular method for accelerated computing. GPUs offer superior performance for tasks with a lot of thread-level parallelism and floating-point calculations. Effort is also put in the more power-efficient FPGA accelerators, suitable for deeply pipelined datapaths and other highly parallel algorithms. In either case, there is little explicit support for accelerated computing in specifications or implementations of major JVMs at the time of writing.

One of the major challenges during integration of JVM programs with accelerators is transferring the data represented as objects in the JVM memory to the accelerator. The interface between the data stored in objects managed by the JVM and an accelerator incurs a specific amount of overhead. If this overhead is higher than the performance gained from the accelerator, there is no point in investing effort to accelerate an algorithm.

Because we look at this matter within the context of big data frameworks, we assume that there is an application that would like to perform a transformation on a parallel collection of data items, represented as JVM objects. One example is a map transformation, which is a common operation for big data applications. The goal of this paper is now to give an overview of four different yet feasible approaches in transferring the object data from the JVM to the accelerator. Furthermore we attempt to quantify the overhead of this data transfer based on the layout of the object and the approach taken. This can help future development and integration of accelerators with JVM-based big data frameworks.

The contributions of this section are:

- We give an overview of the most feasible approaches in transferring data between JVM and accelerator.

- We present a benchmarking tool that quantifies the serialization throughput for a given platform. This allows designers and researchers to get an estimation of the performance of their accelerated implementation, when JVM objects hold the source data. It will also give an indication on which approach will suit their performance requirements. The tool can also be used as a static analysis tool on custom object layouts.

- We measure the data transfer performance of each of the approaches on the OpenJDK HotSpot VM running on a POWER8 system.

The organization of the section is as follows. Section 2.2.1 will discuss related work. In Section 2.2.2 a more thorough problem definition is given. In Section 2.2.3 we will give an overview of four feasible approaches. Section 2.2.4 comprises the experimental setup. In Section 2.2.5 the results are presented. A conclusion is given in Section 2.2.6.

2.2.1. RELATED WORK

When accelerators are controlled by and attached to a host system, it is assumed that the accelerator interface partially consists of a native library. Therefore, the problem of transferring object data from JVM to accelerator is initially similar to the problem of native function access to JVM memory.

For this reason, the JVM implements the Java Native Interface (JNI). Many open-source projects exist (e.g. JNA [13], JavaCPP [14]) that mainly attempt to simplify integration of native libraries with Java programs through sugaring or abstraction of the JNI (which is normally used by writing C or C++ code). We will measure the best case performance of JNI without any of the overhead introduced by the frameworks.

Several researchers that attempt to integrate accelerators with JVM based big data frameworks note that the JNI interface causes a major performance bottleneck for their applications. In the work of [15][16], the massive latency that the JNI introduces is hidden by task pipelining, effectively overlapping JNI access with accelerator execution. Also, when a succession of transformations will take place on the accelerator, intermediate data is cached in local memory and can also be broadcast to other nodes as is.

Another interesting scheme is described in [17], which uses direct ByteBuffers for which the backing array is mapped to a region accessible through direct memory access by an FPGA. The authors use the Xilinx Zynq system, in which the host CPU shares the same physical memory as the FPGA. Such hardware setup enables easy access to the data from the accelerators, although it is not yet common in today's big data clusters.

In the Apache Spark project, with the introduction of the Tungsten engine, data items can be stored in off-heap memory using DataFrames and Datasets. At the time of writing, this is currently done mainly to prevent garbage collection overhead on the large collections of data. The community has been discussing the possibility of feeding data that is stored off-heap directly to native libraries, but any design, implementation or measurements have not yet been presented [18]. Furthermore, objects are serialized into the off-heap memory using serializers that in many cases will also introduce extra overhead by e.g. compressing the data [19].

More recently, with TensorFrames [20], integration of GPGPU accelerated computing in Spark using off-heap managed data is shown. However, when analysing the implementation in early 2017, internally the data is first deserialized back into the JVM and then JavaCPP is used (which is JNI based) to transfer the data row-wise through a native library to a GPGPU, which is rather inefficient. In the work on Spark-GPU [21], a similar approach is seen. Data items first have to be transferred to some off-heap memory region, before passing it to a GPGPU. In a specific case with string objects, the authors show that reading back this data from a GPU-friendly projection in an off-heap structure incurs significant overhead of between 10.5× to 18.3×.

Many of these previous works focus on accelerating a specific application, in which the interface between JVM and accelerator is not the main point of thorough investigation. In this work, we aim to give an overview and quantify *in detail* the properties of this interface, since it is one of the most critical components in such a system.

2.2.2. Problem definition

In light of the advancing interest in accelerating JVM based big data applications and frameworks, the main question that this section aims to address is as follows. *Which approaches exist to transfer data held by objects in a JVM to an accelerator, and how efficient are they?*

To scope the question, we assume that there is an application holding a collection of objects that contain data of interest to be used in a transformation. The transformation is implemented in an accelerator. This commonly means that each object in the collection will be transformed to some new object, or it will be reduced to some final result. We also assume the collection is parallel, thus the transformation may be applied to the objects in parallel as well, i.e. the objects within a collection do not refer to each other.

The fields of an object that represent its state and data, can be of the following types:

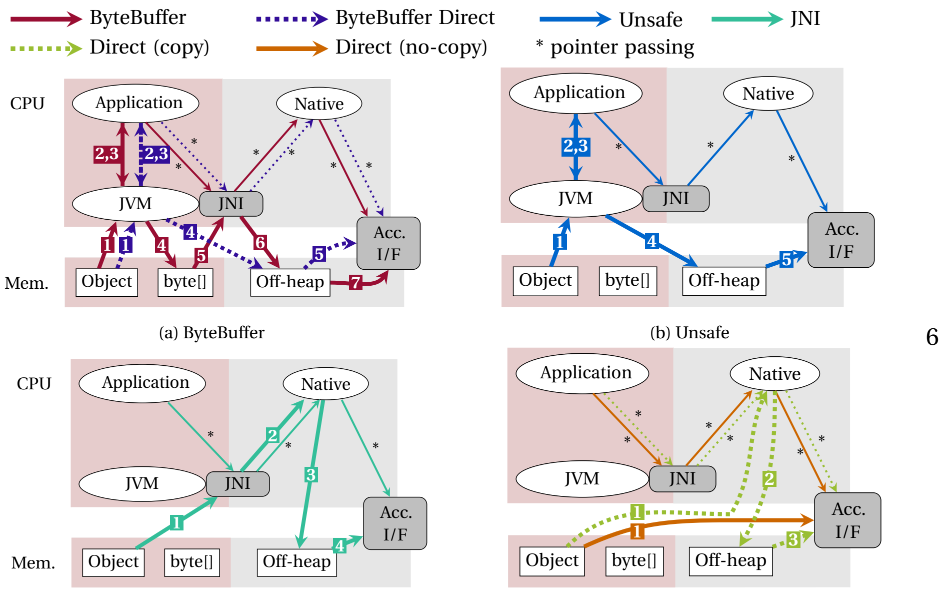
- A primitive (e.g. an integer, float or character).
- A reference to a child object or a child array object.

One exception is the array object type; it can hold multiple primitives or references, where each primitive or reference does not have a separate field identifier, merely an index. For the sake of simplicity we will assume that there are no loops in the reference graph of the object, i.e. all object reference graphs are trees.

The main problem within this scope is due to the fact that a programmer running applications on a JVM has no explicit control over the location or the layout of the objects in memory. In a system where one has control over both layout and location of objects, one may choose to lay out the data in such a way that it is convenient for an accelerator interface to access. This usually means that the data at least resides in a memory region that is as contiguous as possible.

Thus, to perform the transformation on one object of the collection, all primitives that reside in the object tree must first be obtained. This involves traversing the object tree, accessing all the primitives, whether they are in fields or in arrays. When this data is collected and saved in a contiguous memory region, this process is also known as *serialization*. Serialization is used to store the object to disk or to transfer it over a network, hence the object must usually be placed in a contiguous memory region so that it may fit in a file or a message. Later on, the serialized object is deserialized into the memory of another JVM. However, for a transformation to take place in an accelerator, the primitives must merely be transported to the accelerator's local memory; not necessarily reconstructing it in such a way that a JVM program can access it again.

Accelerators are often controlled by a host CPU. In most cases this CPU will also run the JVM. Controlling the accelerator from the application will therefore involve calling at least one, but possibly multiple native functions. In major JVM implementations, this can be done using the Java Native Interface (JNI). Therefore, there are two ways of where object traversal could take place; either by bytecode running on the JVM itself or by a native function invoked through the JNI.



(a) ByteBuffer

(b) Unsafe

(c) JNI

(d) Direct

Figure 2.3: Four different approaches of transferring the object data to the native environment. The thick lines represent the data path, and pointer/reference passing between different programs is shown with an asterisk. Label numbers indicate data flow order.

To address the question posed at the beginning of this section, the next section will give an overview of four feasible approaches to obtain the primitives and transfer them to an accelerator.

2.2.3. OVERVIEW OF DATA TRANSFER APPROACHES

This section first discusses the approaches of accessing a single JVM object using a single thread, and then how multiple objects can be accessed using multiple threads.

SINGLE-OBJECT, SINGLE-THREAD APPROACHES

There are four basic approaches by which the data of an object could be transferred to an accelerator (also shown in Fig. 2.3), namely:

a) *ByteBuffer* approach — Using the JVM to traverse the object tree and write it into a byte array using the `java.nio.ByteBuffer` or its derivatives like `IntBuffer` or `FloatBuffer`. The byte array is then passed to the accelerator interface through the JNI.

b) *Unsafe* approach — Using the JVM to traverse the object tree and write it directly to off-heap memory using the `sun.misc.Unsafe` library. The address of off-heap memory location of the object is then passed to the accelerator interface.

c) JNI approach — The object reference is passed as an argument through the JNI to a

Table 2.1: Summary of characteristics of different approaches

Approach	**Traversal**	**Serialized**	**Copies**	**Portability**	**Support**	**Seen in**
ByteBuffer	bytecode	yes	1-2	high	high	[19]
ByteBuffer (off-heap)	bytecode	yes	1	high	high	[17]
Unsafe	bytecode	yes	1	medium	low	[20][21]
JNI	native	yes	1-2	medium	high	[13][14][22]
Direct (copy)	native	yes	1	low	low	—
Direct (no-copy)	native	no	0	low	low	—

native function. Then, the native function uses JNI functions such as `Get<Primitive>Field` or `Get<Primitive>ArrayElements` to obtain the primitives.

d) *Direct* approach—Traversing the object tree directly while it resides inside the JVM memory. The accelerator interface may directly load the data or it may first be serialized in some off-heap memory location.

The following subsections will discuss each approach in more detail. A summary can be found in Table 2.1.

ByteBuffer approach: For this approach, the object tree is traversed using JVM bytecode. Primitives and primitive arrays are copied to a `ByteBuffer` using its `put` and `get` methods. ByteBuffers are objects that wrap around a byte array (called the *backing array*). They allow an easy interface to load and store primitives from and to the byte array. The reference to this byte array can be passed through the JNI to a native function that interfaces with the accelerator. To obtain the actual array, the JNI function `Get<Primitive>ArrayElements` or `GetPrimitiveArrayCritical` can be used. This may[1] cause another copy (although less likely in the latter case) of the data, but in either case it makes the array accessible to the native function. A variant to this approach is where the ByteBuffers can also wrap around an off-heap byte array, if the byte array is allocated using the `allocateDirect` method. In this case, the data only has to be copied from VM memory to the byte array once. The address of the off-heap byte array can be obtained in the native code by using the JNI function `GetDirectBufferAddress`.

Unsafe approach: The Unsafe approach uses the `sun.misc.Unsafe` library. This library allows C-like memory operations such as allocation and freeing of off-heap memory. Within the Java programming paradigm it is considered ‘unsafe’ because usually memory management is not done explicitly by the programmer. The library is tightly coupled with the HotSpot VM, but its interface is not officially supported or standardized. Traversal of the tree is done using JVM bytecode. Primitives and primitive arrays are copied to an off-heap allocated memory location using `put` and `get` methods. This makes the Unsafe approach quite similar to the ByteBuffer Direct approach. To access the data, the memory

[1] this depends on whether the representation of the array in the VM is the same as the native representation, and if the VM garbage collector supports "pinning"

address of the off-heap structure can be simply passed to a native function interfacing with the accelerator using the JNI.

JNI approach: The JNI approach is less straightforward, since traversing the object tree is done through JNI calls in the native code. First, the references to the classes of the objects in a tree must be obtained using `FindClass`. Then, the field IDs of the classes must be obtained using `GetFieldID`. Object references can be traversed using `GetObjectField`. Finally, with `Get<Primitive>Field`, primitive fields can be obtained. The functions `Get<Primitive>ArrayElements` or `GetPrimitiveArrayCritical` can be used to obtain array values, where both functions potentially copy the values into a newly allocated region that must be released afterwards. Because an accelerator cannot call JNI functions directly, it is assumed that when the JNI approach is used, the primitives are stored in some memory allocated by the native code, and thus the object is serialized. The serialized object is then passed to the accelerator interface.

Direct approach: The Direct approach involves traversing the object tree and obtaining the values from the JVM memory itself. Traversing the object tree is done through calculating pointers to the objects from JVM references directly. Fields are taken from offsets on the object pointers. This approach has low portability since the way in which references are represented and translated to virtual memory addresses is not standardized across implementations. For example, in the HotSpot VM, this depends on VM parameters and platform address size. References (called *ordinary object pointers* or OOPs) can be 32- or 64-bits, where the 64-bit representation is an actual native pointer, but the 32-bits representation might be a compressed OOP [23]. Also, the offsets or implementation of field storage is not specified. Therefore, this approach is not straightforward and is extremely platform-dependent.

If the accelerator has an interface that allows to initiate loads/stores from/to the host application memory that is running the JVM (e.g. CAPI [24], or with CPU + FPGA SoCs where the acceleration fabric shares the data bus of the CPU [17], or with techniques such as NVIDIA's Unified Memory in CUDA), it is not required for the object to be serialized. Instead, the actual object traversal may take place on the accelerator itself. Therefore, the Direct approach allows serialized (copy) but also unserialized (no-copy) access to the object, which is unique to this approach.

We have not found an accelerator interface leveraging this technique published in literature. Note that for server-grade systems this technique seems yet unfeasible, since the latency of contemporary accelerator interfaces is still in the order of microseconds. When reference are traversed in large data structures with small objects, this will result in poor performance, because the ratio of requests to data is high.

It might appear that proper functioning of this approach can be endangered by the JVM garbage collection mechanism. However, when starting the Direct approach through a single JNI invocation that uses an object reference as a parameter, this reference is made a local reference. This means that as long as the JNI function has not yet returned, the garbage collector will not move the object of this reference, or its children. The reference could even be made global such that the object will not be garbage collected at all and can be passed between different JNI invocations or threads.

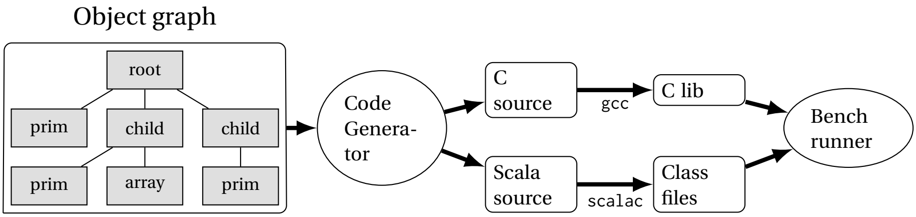


Figure 2.4: General flow of the benchmarking tool

PARALLEL ACCESS OF COLLECTIONS OF OBJECTS

When there is a collection of objects to be processed by an accelerator, and the objects of the collection do not refer to other objects in the collection, the collection may also be accessed in parallel. This can help to increase the throughput on multicore systems. In case of the Direct approach with load/store capable accelerators, loads for data of multiple objects could be pipelined.

To support parallel access for the ByteBuffer approaches, each thread gets its own ByteBuffer object with backing array in order to prevent race conditions. The backing arrays are obtained through the JNI and could be merged in the native code. They could also be sent to the accelerator interface in sequence, thus from the accelerator point of view, it will no longer be a single collection per se. This might introduce some overhead or require a slightly more complex control structure for the ByteBuffer approach.

For the Unsafe approach, the memory is allocated once, then each thread gets its own instance of `sun.misc.Unsafe`, again to prevent race conditions. Then, each thread operates on a different offset of the destination memory.

For the JNI approach, parallel access is more complicated. Because references to objects are only valid in the corresponding thread that obtained them through the JNI, the reference to the array holding the collection must first be made global before it is passed to each thread. New threads must also register with the JVM using the JNI function `AttachCurrentThread` before they may call other JNI functions. After accessing the values, the threads must detach and the global reference must be released to allow garbage collection to take place on the objects.

For the Direct approach, parallel access is straightforward. Multiple threads may have multiple outstanding accesses of JVM memory simultaneously, and store them on different offsets of off-heap memory.

2.2.4. EXPERIMENTAL SETUP

To measure the access times of the different approaches, a benchmarking tool was implemented (see also Fig. 2.4). As an input to the tool, a layout of an object tree is first specified. From this specification, Scala sources and ultimately JVM bytecode is generated, containing the required object classes and an `Instantiator` class which contains methods to instantiate the object tree and fill it with random data. Furthermore, for each approach, in the top-level class, methods are generated to serialize the object to a byte array or to off-heap memory corresponding to the description of the ByteBuffer and Unsafe approach, respectively. These classes are then compiled to JVM bytecode. For the

JNI and Direct approach, functions callable through the JNI are generated in C. They are then also compiled to a shared library, together with functions that access the data for the ByteBuffer and Unsafe approach. Finally, a second program (benchmark runner) can take the class files and library as an input and run measurements of object access time. The benchmarking tool is available as an open-source project[25].

Using this tool, it is possible to measure the access times of different types and sizes of object trees. It is possible to generate two types of object layouts. One layout where the root object does not contain any references, except to primitive arrays (a *leaf* object). A leaf object contains only a variable amount of primitives and arrays of variable size. Another layout where the object has a specified width W and depth D, such that at the root level it contains W references to the second level, where each object contains $max(W-1,1)$ references, until the level equals D. Only at the last level, the primitives and arrays are instantiated. This layout allows us to also make a linear object tree, by setting $W = 1$ and $D > 1$. It is also possible to supply a custom class layout or object tree to the tool, such that it may be used as a static analysis tool for existing applications.

By varying the parameters of the aforementioned object tree layouts, it is possible to obtain the serialization throughput for a specific layout, and a specific platform and JVM implementation. By varying the number of primitives in a leaf object, the average time to copy a primitive can be measured. By increasing the array size of a leaf object, the average array copy time per element can be measured. By varying the depth in a linear object tree, the average time to traverse a reference can be measured.

Parallel collection serialization and parallel access performance is also measured to get the peak performance for the platform. From the hardware point of view, when more cores and hardware threads run in parallel, we may assume that the internal memory infrastructure is at some point saturated, resulting in peak performance for that platform. Thus, besides specifying the object tree layout for a benchmark, it is also possible to generate a collection of N of these objects and specify with how many threads to access the objects. Furthermore, because run-time measurements on the JVM are very noisy, the experiments are repeated R times and averaged. These numbers are reported per experiment in Section 2.2.5.

The benchmarking system consists of two POWER8 CPUs running at 3.42 GHz on an IBM Power System S824L (8247-42L) with 256 GiB of total RAM. We confine our measurements to one of the two CPUs only. Primitives used are 32-bit integers. Attaching an actual accelerator is outside the scope of this paper, but by accumulating all serialized primitives into a single value on the CPUs we can validate the correctness of each approach and we can make a fair comparison for the Direct (no-copy) case. This is in theory the fastest approach, because it does not copy at all. Without any operations on the data it would otherwise only have to resolve all references without accessing the data itself. Native threads are controlled through OpenMP and JVM threads are statically managed.

In a real application, more dimensions to the problem are relevant, such as how to lay out the serialized objects in the memory such that its structure is convenient to process in a specific accelerator, how to retain references within the serialized format, how to deal with cyclic object graphs, how to deal with static fields of classes, and more detailed problems which are commonly seen in serialization. However, because we are primarily interested in the feasibility and best-case performance, these dimensions are also outside the scope of this work.

2.2.5. Results

Single object

In Fig. 2.5a, the access time of a leaf object with an increasing number of primitive fields is shown. We found that the ByteBuffer, Unsafe and Direct approaches show a mainly linear increase in access time, while the JNI approach shows a significant quadratic increase when the number of fields increases in a leaf object. This is due to the use of the JNI function `GetFieldID` for this particular experiment. This function looks up the field identifier string in the HotSpot symbol table. Suppose the number of primitives is p. We must search p symbols p times to access all primitives. Thus the time complexity to access a field by using `GetFieldID` is $O(p^2)$.

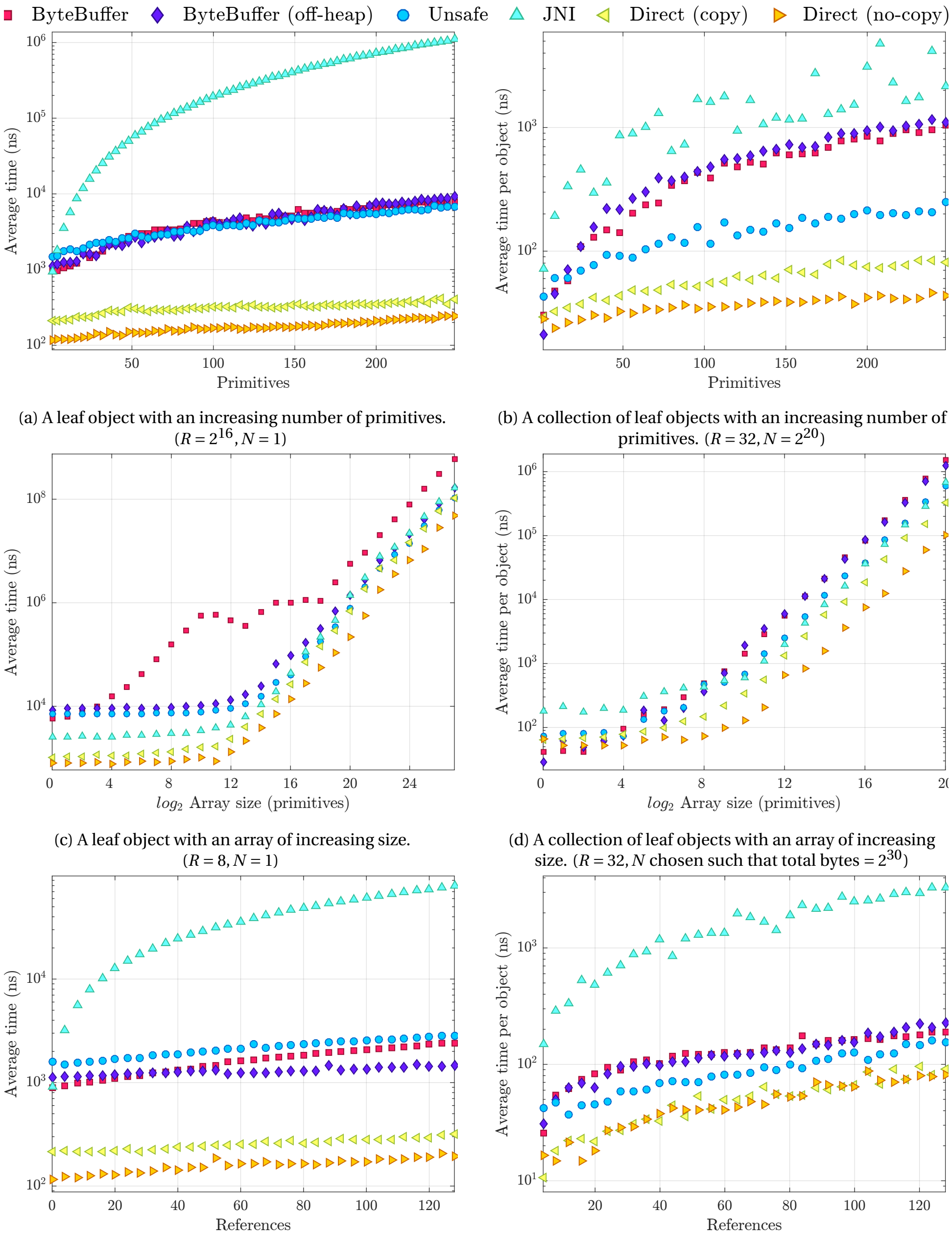
In Fig. 2.5c the access time of a leaf object containing only an array is shown. Accessing arrays of different sizes clearly shows the effect of the CPU cache hierarchy. The derivative of the access time with respect to the array size is small in regions where the array still fits in the cache. It becomes larger for array sizes that do not fit in the cache. Any initial overhead of the copies becomes relatively small. For each approach that has the same number of copies (see Table 2.1), for large arrays, their access times converge, because internally array copies are usually performed by highly optimized `memcpy` calls (although variants depending on the native platform exist, e.g. for the Unsafe and ByteBuffer approaches).

In Fig. 2.5e, the access time of a linear object tree is shown. We found that the access times increases in a mainly linear manner with respect to the number of references traversed.

Parallel performance

In the case of a parallel collection, we first attempt to find a suitable number of threads. For each approach we measuring three cases; 1. small objects (2 primitives and an array with 16 primitives) 2. medium objects (8 primitives and an array of 1024 primitives) and 3. large objects (64 primitives and an array of 800×600 primitives. Reference traversal performance is included in these measurements since a collection consists of many references to all its objects. The collections are of such a size that their serialized representation is over several hundred MiB, to make sure each thread has enough work to justify the overhead from spawning it. The results of these three measurements are seen in Fig. 2.6.

From these measurements, we found that for all approaches except the Direct approaches, the scalability is rather poor. This is most likely due to race conditions on specific resources of the VM. These approaches scale very badly when the ratio of reference to data is high (small objects case). In the case of a high ratio of reference to data, the maximum number of threads even gives the best performance for the Direct approaches. In the medium and large object cases, the speedup increases all the way to the maximum number of threads only for the Direct (no-copy) case. This approach only performs a single load and accumulate on each data item. The gains from multi-threading are therefore more significant than in the case of the Direct (copy) approach, because the computation to communication ratio is three times higher. The Direct (copy) approach loads, stores



(a) A leaf object with an increasing number of primitives. ($R = 2^{16}, N = 1$)

(b) A collection of leaf objects with an increasing number of primitives. ($R = 32, N = 2^{20}$)

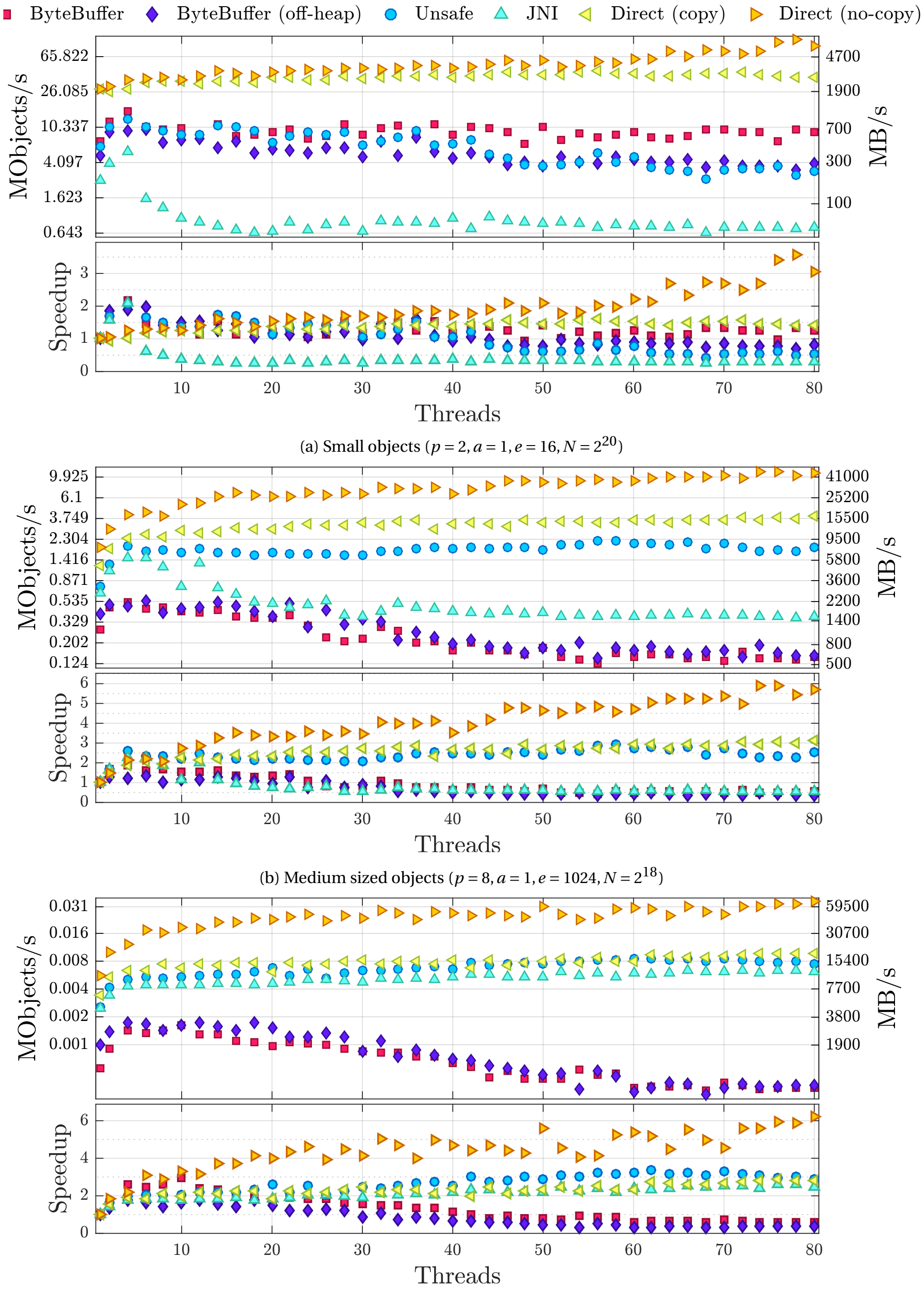
(c) A leaf object with an array of increasing size. ($R = 8, N = 1$)

(d) A collection of leaf objects with an array of increasing size. ($R = 32$, N chosen such that total bytes $= 2^{30}$)

(e) An object with a linear object tree with increasing depth. ($R = 2^{16}, N = 1$)

(f) A collection of objects with a linear object tree of increasing depth. ($R = 4, N = 2^{21}$)

Figure 2.5: Average latency of accessing JVM objects.



(a) Small objects ($p = 2, a = 1, e = 16, N = 2^{20}$)

(b) Medium sized objects ($p = 8, a = 1, e = 1024, N = 2^{18}$)

(c) Large objects ($p = 64, a = 1, e = 800 \times 600, N = 2^{10}$).

Figure 2.6: Throughput and speedup of accessing collections of JVM objects.

Table 2.2: Maximum throughput and corresponding number of threads for the small, medium and large benchmark.

Approach	Small (threads)	MB/s	Medium (threads)	MB/s	Large (threads)	MB/s	Threads used for Fig. 2.5
ByteBuffer	4	1142	4	2159	10	3121	4
ByteBuffer (off-heap)	6	706	6	2231	4	3346	4
Unsafe	4	914	58	9080	62	16604	10
JNI	4	389	6	6094	78	12332	4
Direct (copy)	56	3232	80	16273	76	18788	80
Direct (no-copy)	78	7366	76	47064	80	67381	80

and then loads again and accumulates the data. The Unsafe approach also scales rather well in the medium and large measurement, compared to the JNI approaches, which scales only well for the large case. The ByteBuffer approach does not scale very well beyond four threads. From these measurements we set a suitable number of threads for each approach as shown in Table 2.2.

COLLECTION

In the case of a parallel collection, the same types of measurements are performed as in the case for a single object, although on a collection of N of these objects. These measurements are shown in Fig. 2.5.

From measurements shown in Fig. 2.5b, we found that the quadratic increase in access time for the JNI approach is now relatively insignificant, because the field identifiers only have to be looked up once for the whole collection. However, the JNI approach is still slow, because for each field primitive, the function call `Get<Primitive>Field` JNI must still be made.

Fig. 2.5d shows that after different initialization times, approaches with the same number of copies converge towards the same access latency as the arrays get larger, as was the case for the single-thread single-object measurements.

Lastly, for the measurement of reference traversal in a linear object graph (Fig. 2.5f), the direct approaches show similar access time, followed by the Unsafe and ByteBuffer approaches Theoretically, there should not be much difference between the Unsafe and ByteBuffer approaches, because reference traversal is done in the JVM in the same way for both approaches. The difference in average time for the ByteBuffer approach is due to the overhead induced by spawning the threads and ByteBuffer objects. For the Unsafe approach, this overhead is much lower. Again, the JNI approach shows an order of magnitude worse performance.

DISCUSSION

Contemporary commercially available accelerator cards are often connected via PCI-e GEN3 with peak bandwidths of almost 8 GB/s or 16 GB/s, depending on the configuration. One example includes the POWER8 system where the CAPI interface can be used over such a link. Newer interfaces such as OpenCAPI and NVIDIA NVLink are expected to achieve up to 25 GB/s and 80 GB/s, respectively.

From the measurements presented in this section, it can be seen that the ByteBuffer approaches are generally unfavorable, because they cannot achieve near the bandwidths of the PCIe range easily. They are only faster than using the JNI approach in the case of accessing a collection of small objects. The JNI approach can be a feasible solution, but only when the ratio of references-to-data is low (e.g. when there are few but large arrays in the objects). At the same time, the Unsafe approach performs better in most cases and it is much easier to program, because traversal of the object graph can be written or generated in the JVM based source language, and calls to these functions may be inlined during JIT compilation of the serialization function. If it is necessary to saturate the link, with small objects (a high reference/data ratio) the only feasible solution is to take the Direct approach.

A major drawback of this approach is that it is highly platform dependent and it could even be considered more 'unsafe' than the Unsafe approach, since it accesses VM managed memory without some sort of interface that was designed into the VM. To mitigate this drawback, the VM could *by design* include some functionality to support fast object graph traversal, serialization and data transfer to accelerator interfaces, implemented with native code and tightly coupled with the VM as in the case of the Unsafe library.

2.2.6. Conclusion

In this section, an overview is given for four different approaches for accelerator interfaces to obtain data from JVM managed objects; using ByteBuffers, the `sun.misc.Unsafe` library, the Java Native Interface (JNI) and to directly obtain the data from JVM managed memory (Direct).

A benchmarking tool was implemented that generates code to serialize the object or a collection of objects for use in an accelerator, using these four different approaches (where two approaches have two variants). By measuring the access times of single objects by a single thread, and access times of a parallel collection of objects by multiple threads, the performance of a POWER8 system with the HotSpot VM was measured. Furthermore, the throughput of a collection of small, medium and large objects was measurement with respect to the number of threads.

From the measurements we may conclude that the ByteBuffer approach does not perform well in most cases (it can achieve between 0.7 and 3.3 GB/s of throughput). Also, it does not scale well with the number of threads. The JNI approach can perform well in situations where the ratio of references to data is low, but also scales poorly with the number of threads (it can achieve between 0.9 and 12 GB/s of throughput). The Unsafe approach scales slightly better, up to the number of physical cores of CPU, and is also able to provide enough bandwidth to saturate common accelerator interfaces such as PCIe gen. 3 (it can achieve between 0.8 and 16 GB/s of throughput) for large objects, although for smaller objects, the performance is also limited. However, newer and upcoming accelerator interfaces, such as OpenCAPI, NVLink, or CXL, cannot yet be saturated through this method, and the scalability is poor. The best approach in terms of performance is the Direct approach. It scales well and offers more than enough bandwidth for common accelerator interfaces, but its portability and ease of use is poor (it can achieve between 3 and 67 GB/s).

The measurements of the benchmarking tool can effectively be used to predict the interface bandwidth of accelerators attached to a JVM. This may help researchers and developers to obtain a good estimation of the maximum speedup they may get by combining accelerators with JVM-based applications.

As new accelerator interfaces with higher bandwidths are introduced, the need for a faster interface that is integrated into the HotSpot VM *by design* is high. This is especially the case if users of big data frameworks based on the JVM want to make use of the computational power of accelerators.

2.3. APACHE ARROW

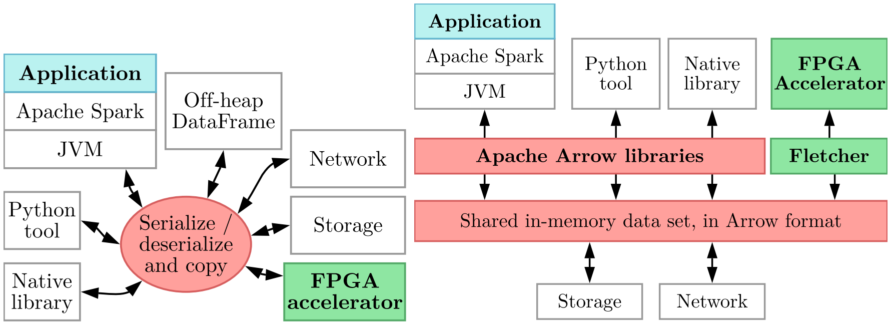
Due to the nature of this dissertation, challenges S1, S2, and S3 from Section 2.1 were described mainly from an FPGA acceleration developer point of view. We have thoroughly quantified the limitations of serialization with respect to accelerator interfaces in the previous section. Serialization is generally an unwanted necessity, as it merely transforms the form rather than the contents of the data, and is therefore a non-functional aspect.

However, even within the software ecosystem of big data analytics pipelines, the challenges related to hardware-unfriendly in-memory layouts and serialization exist. When heterogeneous processes interact (e.g. when there is inter-process communication between a pure Python program off-loading some computation to a very fast C library), there needs to be one common (in-memory) format that both programs agree on. Several projects have provided such a common format for generic types of data, such as Google's Protobuf [26]. The project provides a code generation step to automatically generate serialization and deserialization functions that help produce and consume data in the common format, turning it back into language-native in-memory objects. In this way, programmers can continue to work with the objects in the fashion of their language.

Later, it was realized that serialization and deserialization itself can cause bottlenecks, since copies have to be made twice; first when serializing the data to the common format at the producer side, and again when deserializing it on the consumer side. In applications built on top of these analytics frameworks, serialization may take up a large portion of the run-time of the full application [27]. Examples of where serialization takes place between components of a heterogeneous framework such as Apache Spark [9] can be seen in Figure 2.7a.

In many cases, providing specialized functions to access the data in its common format is faster than applying serialization and deserialization, since data may be passed between processes without making any copies to restructure it into a language-specific format. This has led to what is called a zero-copy approach to inter-process communication. Through the help of libraries such as Flatbuffers [28], such functions are provided to several languages. Producing processes immediately use the common format for their data structure, and then only share a pointer to the data with the consuming process. No copies are made because both processes work with the common format as much as possible from the same location in memory. Programmers are provided with language-specific libraries that make it easy for them to interact with the data structure according to the fashion of their language.

An approach similar to Flatbuffers, but specifically tailored to big data analytics on structured data, is found in the Apache Arrow project [29]. Apache Arrow is specifically tailored to work with large tabular data structures that are stored in memory in a column-



(a) Examples of where serialization can take place in a typical big data system.

(b) How Arrow attempts to prevent serialization through the use of a common data layer. Fletcher is a contribution of this dissertation based on Apache Arrow.

oriented fashion. While iterating over column entries in tables, the columnar format allows more efficient use of CPU caches and vector instructions than a row-oriented format. It also provides a memory management daemon utility called Plasma, that allows to place the data structures outside the heaps of garbage collected run-time systems, furthermore providing interfaces for zero-copy inter-process communication of Arrow data sets.

Thus, Arrow specifically deals with the challenges S1, S2, and S3 by, respectively:

1. Allowing data to be stored off-heap, unburdened by GC.
2. Providing a common in-memory format and language-specific libraries to access the data, preventing the need for serialization.
3. Tailoring the format to work well on modern CPUs, by being column-oriented and as contiguous as possible.

As we have discussed in Section 2.1, the last point is especially important when moving large amounts of data over relatively 'long' distances, e.g. over an accelerator interface between host memory and accelerator on-board memory. When the data resides in large contiguous buffers, we may transport the data with large bursts, increasing the throughput by relatively decreasing the overhead of initiating new transfers. The Arrow in-memory format dictates that data buffers must be very contiguous compared to commonly used, language-specific container types. This is also shown in Figure 2.2d, on which we will elaborate in the rest of this section.

2.3.1. IN-MEMORY FORMAT

Arrow data sets are typically tabular and stored in an abstraction called a RecordBatch. A RecordBatch is accompanied by meta-data called a *schema* that specifies the types of the fields of the objects/records stored in the table. A RecordBatch contains several columns for each field of a record, that are in Arrow called *arrays*. These arrays can hold all sorts of data types, from strings to lists of integers, to lists of lists of time-stamps, and others[2].

[2] Therefore, Arrow Arrays are not to be confused with C-like arrays of fixed-size elements.

(a) Schema:

Field A:
Float (nullable)
Field B:
List(Char)
Field C:
Struct(E: Int16, F: Double)

(b) RecordBatch:

A	B	C
0.5f	"fpga"	(42, 0.125)
0.25f	"fun"	(1337, 0.0)
∅	"!"	(13, 2.7)

(c) Arrow buffers:

Index	Buffers for:					
	Field A		Field B		Field C	
	Validity (bit)	Values (float)	Offsets (int32)	Values (char)	Values E (int16)	Values F (double)
0	1	0.5f	0	f	42	0.125
1	1	0.25f	4	p	1337	0.0
2	0	×	7	g	13	2.7
3			8	a		
4				f		
5				u		
6				n		
7				!		

Figure 2.8: An example schema (a) of a RecordBatch (b) and resulting Arrow buffers (c).

Arrow arrays consist of several contiguous Arrow *buffers*, that are related, to store the data of a specific type. There are several types of buffers. In this work we consider *validity* buffers, *value* buffers and *offset* buffers.

Validity buffers store a single bit to signify if a record (or deeper nested) element is valid or *null* (i.e. there is no data). Value buffers store actual values of fixed-width types, similar to C arrays. Offset buffers store offsets of variable length types, such as strings (which are lists of characters), where an offset at some index points to where a variable-length item starts in another buffer.

When a user wants to obtain (a subset of) a record from the RecordBatch, through the schema, we may find out what buffers to load data from to obtain the records of interest. An example of a *schema*, a corresponding *RecordBatch* (with three *arrays* and the resulting *buffers* are seen in Figure 2.8.

2.3.2. FPGA INTEGRATION CONSIDERATIONS

Typically, an FPGA accelerator developer designs an accelerator kernel that has to request access to the data sets through a byte-addressable memory interface. That means the accelerator must typically request a bus word from a specific byte address. In relation to the previously mentioned challenge H2, however, in the case of a tabular data set stored in the Arrow format, it would be more convenient to express access to the data by supplying a table index, or a range of table indices, and receiving streams of the data of interest in the form of the types expressed through the schema, rather than as a bus word. This is illustrated for the example of the strings in Figure 2.2e. Because schemas allow the expression of a virtually infinite number of types through nested combinations, an implementation of such a mechanism is challenging, and will be discussed thoroughly in Chapter 3.

The fact that there is a higher level description of the data structure (the schema) furthermore provides an advantage. While designing the functional aspects of an FPGA accelerator can already be challenging, a significant portion of design time involves structural aspects of the infrastructure feeding such interfaces, as described in challenge H3. Interface design often deals with converting data on very wide hardware buses (the datacenter-grade platforms used in this dissertation use between 512 and 1024 bits) to something more usable at the input of the accelerator. This includes pointer arithmetic to determine which bytes are the bytes of interest, parallelizing or serializing words into larger or smaller chunks, and shifting them into the right positions before turning them into data streams to be absorbed by some kernel.

The relation between the raw bytes of a RecordBatch are known from the schema and the format specification. It is therefore possible to automatically generate circuits that perform the required pointer arithmetic and pre-processing of raw bus words into streams that are more meaningful and usable to an accelerator developer. More specifically, based on the schema, an interface may be generated that as a command takes a range of object/records indices of a RecordBatch and streams out the requested fields as exactly the data types expressed in the schema. Furthermore, parts of the control and data flow on the host-side may also be automated (e.g. passing buffers addresses and potentially moving data to accelerator on-board memory).

With such a setup, would it possible to operate at system bandwidth using the Apache Arrow in-memory format? In general, any serialized format suitable for FPGA processing causes as few pointer traversals as possible, requires as little pre-processing or reordering in the accelerator as possible and is streamable. With this in mind, we investigate two forms of data that can be generalized to all data structures; fixed-width data fields and variable-length data fields.

Fixed-width fields

RecordBatch columns with fixed-width elements (e.g. `floats`, `booleans` or `ints`) are in Arrow format stored in one contiguous *values buffer*, equivalent to a C-like buffer. Given some index of data to obtain, an offset has to be calculated, the specific data word (or words) have to be loaded. Upon receiving the raw bytes, the bus words have to be shaped into the correct type, before they can be presented on a streaming output. If kernels can absorb multiple elements per cycle, or if multiple kernels want to read from the same column in parallel, it is possible to match system bandwidth on such an interface. Assuming a kernel requests the full range of objects from the table, only one “pointer” is traversed to read this field for all objects of interest with maximum size pipelined bursts on the memory interface.

This is much more efficient than if the accelerator would have to traverse a pointer for each fixed-width element. For a C programmer it may seem far fetched for a collection of integers to be stored as a list of pointers to integers. However, some high-level languages (such as Python and R) box every integer into an object (hence the need for e.g. *Numpy* providing more efficient, native implementations to perform matrix operations on large collections of integers, among others)). Any interface dealing with such an in-memory lay-out will quickly be bounded by memory latency if such a collection of integers is to be traversed through pointers to the integer objects.

Variable-length fields

More interesting are Arrow columns of variable length types (e.g. a UTF-8 string). They are referred to as lists of some other type (e.g. a `List<Char>` or `List<List<Int> >`). They contain at least two buffers, an *offsets buffer* and the values buffer. An offset at some index in the offsets buffer corresponds to the index of the first element of the list in the values buffer. The values buffer contiguously holds all primitive list elements. This format offers some advantages that an interface generation framework may exploit over what HLS-compilers can assume about this data structure.

More formally, consider the case where a variable length object is represented through two Arrow buffers; the offsets buffer $O = \{o_1, o_2, ..., o_N\} \in \mathbb{Z}^{\geq}$ and values buffer $V = \{v_1, v_2, ..., v_M\}$. O, in the C-language, will be represented as an unsigned integer array. A C-based HLS compiler may not make assumptions about the values of o_i, as they are defined during run-time. More specifically, it cannot assume that in the case of an Arrow offsets buffer, $o_{i+1} - o_i \in \mathbb{Z}^{\geq}$; the outcome of this calculation might also yield a negative integer. Therefore, not to lose generality it must request each run of value buffer elements $v_{o_i}...v_{o_{i+1}}$ separately, and any data path consuming the data is subject to memory latency.

Hardware pre-fetching (such as explored in [30]) or using spatial locality in caches may improve this behavior, but these constructs are costly, especially when, in the case of the Arrow format, they are not required. To elaborate, when requesting a range $j...k$ of variable length objects, in fact the whole range of values of interest $v_{o_j}...v_{o_{k+1}}$ can be requested from the contiguous buffer. This can be bursted into a FIFO, ready to be delivered on the output stream synchronized with a length stream resulting from subtracting two consecutive offsets. Thus, memory latency for pointer traversal is only paid three times independent of the amount of variable length objects that are requested; once to obtain o_{k+1} from the offsets buffer, once to obtain all offsets of interest $o_j...o_{k+1}$, and once to obtain all values of interest. No dynamic hardware pre-fetching or caches are required to deliver throughput that is close to system bandwidth. This approach also generalizes to nested lists.

Furthermore, with these assumptions, this interface can be generated automatically, without the need to manually write an HDL-based interface or the need to write special HLS functions that mimic this optimal behavior. HLS templates for transformation functions used in higher-order functions such as map, filter and reduce, can immediately be provided with length stream and value stream as arguments. Again, this approach generalizes to nested types.

Arrow also supports other convenient data types such as *structs*, sparse and dense unions and dictionaries, which are discussed in its format specification. Furthermore, a special type of fixed-width field that contains a validity bit to allow entries to be nullable is supported.

2.3.3. Limitations

Some limitations to the Arrow approach exist. First, once data sets have been built in memory, it is not trivial to mutate them without breaking contiguousness. Therefore, Arrow is best at storing immutable data sets in memory but less powerful when working with algorithms that aim to mutate data sets in place.

Second, at the time of writing, no data format is specified for graph-based data sets, or other more exotic non-tabular formats. Still, graphs can generally be represented through tables, although there is, at the time of writing, no Arrow standard specification.

A final limitation is that because a different in-memory format is used than some language run-time is used to, code that accesses data (accessors) must go through an additional layer (e.g. some Arrow language specific library) rather than being able to use default ways of accessing object or record fields. While investigating this drawback, we did not find any significant performance degradation. We have investigated C++ (a case where code is compiled to native instructions), where the performance of accessing Arrow based containers is similar and sometimes faster than accessing STL containers, as Arrow exposes raw pointers to the data buffers. For Java (a case where code is compiled to virtual machine bytecode), access to Arrow based data is done through calls to the Unsafe library, as the data is stored outside the VM managed heap. Fortunately, widely-used implementations of the JVM inline these calls during JIT compilation, providing similar performance to normal object field accessors. In Python (a case where code is interpreted), it is common for high performance libraries to use native code underneath (e.g. NumPy) written in Cython. This involves extra developer effort but is a common trade-off made in the Python ecosystem.

Establishing that besides these limitations, the Arrow in-memory format is indeed suitable since it is highly contiguous and streamable, the next Chapter will discuss the implementation of an interface generation framework called Fletcher, based on the Arrow format. As shown in Figure 2.7b, this will add methods to supply FPGA accelerators to operate on data sets in that format.

2.4. Conclusion

In this chapter, we have discussed challenges for FPGA accelerators to become widespread alternatives to existing computational solutions in the domain of big data analytics. We have stipulated three challenges from the software integration side; complex run-time system, hardware-unfriendly in-memory layouts of data sets, and (de)serialization overhead. On the side of designing FPGA accelerators, we discussed three challenges as well; a relative lack of accelerator platform-agnostic open-source tooling geared towards an HDL style flow, a high design effort because of a lack of interfaces tailored towards the data structure needing to be accessed, and a high design effort because the need to design a large amount of infrastructure.

We have quantitatively studied the widely used run-time system, the Java Virtual Machine (specifically the OpenJDK HotSpot VM), with respect to attainable serialization throughput. We have demonstrated that the attainable throughput does not match the bandwidth of contemporary and upcoming accelerator interfaces.

Alternatives from the open source big data systems software ecosystem to deal with the problems of complex run-time systems, hardware-unfriendly in memory layout, and deserialization overhead, have come into existence alongside this dissertation, specifically the Apache Arrow project. We have analyzed the merits of its in-memory format towards solving the challenges on the side of FPGA accelerators as well, and have established that it is a highly suitable format to transport large tabular data structures over accelerator interfaces due to its property of being highly contiguous and hardware-friendly.

References

[1] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, *A cloud-scale acceleration architec-*

ture, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016) pp. 1–13.

[2] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, *A survey and evaluation of fpga high-level synthesis tools,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**, 1591 (2016).

[3] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, *Are we there yet? a study on the state of high-level synthesis,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**, 898 (2019).

[4] J. L. Hennessy and D. A. Patterson, *A new golden age for computer architecture,* Commun. ACM **62**, 48–60 (2019).

[5] L. Truong and P. Hanrahan, *A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity,* in *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136, edited by B. S. Lerner, R. Bodík, and S. Krishnamurthi (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019) pp. 7:1–7:21.

[6] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition* (Oracle, 2015).

[7] I. O. for Standardization, *ISO/IEC 14882:2017, Programming languages–C++,* ISO/IEC **14882** (2017).

[8] J. Peltenburg, A. Hesam, and Z. Al-Ars, *Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?* in *High Performance Computing*, edited by J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf (Springer International Publishing, Cham, 2017) pp. 220–236.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,* in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (USENIX Association, 2012) pp. 2–2.

[10] I. Gouy, *The computer language benchmarks game,* (2017).

[11] M. Anderson, S. Smith, N. Sundaram, M. Capota, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, *Bridging the gap between HPC and big data frameworks,* Proceedings of the VLDB Endowment **10** (2017).

[12] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, *Dark silicon and the end of multicore scaling,* in *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (2011) pp. 365–376.

[13] Open-source project, *Java Native Access,* (2017).

[14] Bytedeco, *JavaCPP,* (2017).

[15] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, *When Apache Spark meets FPGAs: a case study for next-generation DNA sequencing acceleration,* in *The 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[16] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, *Programming and runtime support to Blaze FPGA accelerator deployment at datacenter scale,* in *Proceedings of the Seventh ACM Symposium on Cloud Computing* (ACM, 2016) pp. 456–469.

[17] E. Ghasemi and P. Chow, *Accelerating Apache Spark big data analysis with FPGAs,* in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2016) pp. 94–94.

[18] P. Weiss, *Off heap memory access for non-jvm libraries,* (2017).

[19] Oracle, *Object serialization stream protocol,* (2017).

[20] Databricks, *TensorFrames: Experimental tensorflow binding for Scala and Apache Spark.* (2017).

[21] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, *Spark-gpu: An accelerated in-memory data processing engine on clusters,* in *2016 IEEE International Conference on Big Data (Big Data)* (2016) pp. 273–283.

[22] Z.-N. Chen, K. Chen, J.-L. Jiang, L.-F. Zhang, S. Wu, Z.-W. Qi, C.-M. Hu, Y.-W. Wu, Y.-Z. Sun, H. Tang, *et al.*, *Evolution of cloud operating system: From technology to ecosystem,* Journal of Computer Science and Technology **32**, 224 (2017).

[23] Oracle, *Java HotSpot virtual machine performance enhancements,* (2017).

[24] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, *CAPI: A coherent accelerator processor interface,* IBM Journal of Research and Development **59**, 7 (2015).

[25] J. Peltenburg, *JVM-to-Accelerator Benchmark Tool,* (2017).

[26] Google Inc., *Protocol buffers,* (2020).

[27] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, *Making Sense of Performance in Data Analytics Frameworks.* in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation* (2015) pp. 293–307.

[28] Google Inc., *Flatbuffers: Memory efficient serialization library,* (2020).

[29] The Apache Software Foundation, *Apache Arrow,* (2018).

[30] T. Chen and G. E. Suh, *Efficient data supply for hardware accelerators with prefetching and access/execute decoupling,* in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016) pp. 1–12.

[31] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, *Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow,* in *Applied Reconfigurable Computing*, edited by C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz (Springer International Publishing, Cham, 2019) pp. 32–47.

[32] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee, *Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow,* in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (2019) pp. 270–277.

[33] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, *Generating high-performance fpga accelerator designs for big data analytics with fletcher and apache arrow (under review),* Journal of Signal Processing Systems (2020).

PERSONAL CONTRIBUTIONS, ACKNOWLEDGEMENTS AND FUNDING.
This chapter is based on four articles. Sections 2.1 and 2.3 are based on two published papers [31][32] and one under review [33]. Section 2.2 is based on one published article [8]. A. Hesam helped to explore the implementation of Apache Spark. J.W. Peltenburg wrote the papers, the software, and performed the experiments.

The authors would like to thank Erik Vermij for his help using the POWER8 system and the Texas Advanced Computing Center and their partners for access to the hardware. The authors would like to sincerely thank all developers of the Apache Arrow project, which is managed by the Apache Software Foundation.

This work was supported by the European Commission in the context of the ARTEMIS project ALMARVI (project #621439).

3

The Fletcher framework

In the previous chapter, we have established that the Apache Arrow columnar in-memory format is a suitable candidate for data exchange at the bandwidth of contemporary and future FPGA accelerator interfaces. Besides the aspect of optimizing data movement between big data systems software and FPGA accelerators, we are also interested to decrease the design time associated with implementing and integrating applications for such accelerators. In this chapter, we propose a hardware interface generation framework based on Apache Arrow, called Fletcher. Fletcher generates specialized easy-to-use, scalable, streaming DMA engines for Arrow data structures. This prevents the need for hardware developers to deal with the required memory interfacing control mechanisms emerging from the complex, nested data types that may reside in the columns of tabular data sets. Fletcher furthermore generates all infrastructural logic required to access to multiple columns and tables, providing the entirety of an accelerator design, excluding the computational kernel implementation. Also, the generated designs are vendor-agnostic and portable across multiple data center grade accelerator cards.

3.1. INTRODUCTION

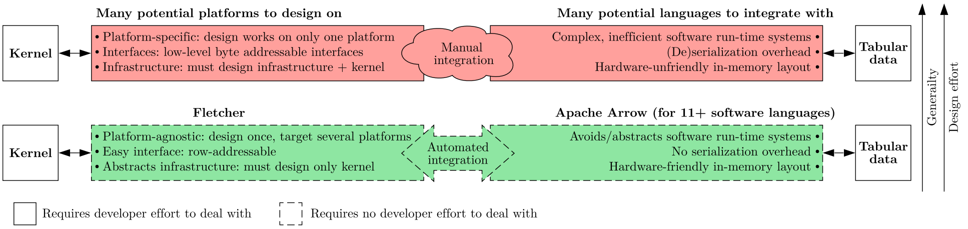

Many potential platforms to design on
Many potential languages to integrate with
Kernel
• Platform-specific: design works on only one platform
• Interfaces: low-level byte addressable interfaces
• Infrastructure: must design infrastructure + kernel
Manual integration
Complex, inefficient software run-time systems •
(De)serialization overhead •
Hardware-unfriendly in-memory layout •
Tabular data
Fletcher
Apache Arrow (for 11+ software languages)
Kernel
• Platform-agnostic: design once, target several platforms
• Easy interface: row-addressable
• Abstracts infrastructure: must design only kernel
Automated integration
Avoids/abstracts software run-time systems •
No serialization overhead •
Hardware-friendly in-memory layout •
Tabular data
Generality
Design effort
Requires developer effort to deal with
Requires no developer effort to deal with

Figure 3.1: The Fletcher FPGA accelerator framework for big data systems, based on Apache Arrow

In terms of both hardware and software, the increasing heterogeneity in (cluster) computing frameworks built for big data analytics causes major challenges [1]. One challenge is that different system components that consume the same data may use different representation of that data in memory. This introduces a serialization requirement whenever data is passed from one component to another, if they are not implemented using the same technology. We have explored this topic thoroughly in the previous chapter, especially in the context of FPGA accelerators.

The Apache Arrow project was launched to (among other contributions) overcome this bottleneck [2], and has already seen integration in several well known tools and frameworks from the data analytics community, such as Dremio, Spark, Dask, and Pandas. The Arrow project defines a common columnar in-memory format for data sets and provides zero-copy inter-process communication libraries for various languages, including (at the time of writing) C, C++, Java, Python, R, Matlab, Go, C#, JavaScript, Ruby and Rust. For a schematic overview, recall Figure 2.7b.

In the previous chapter, we have established that the Apache Arrow format is usable in the context of FPGA acceleration, where serialization bottlenecks can also be present. With the help of Arrow, we can tremendously improve end-to-end accelerated application throughput, because host-side serialization throughput from various high-level languages can generally be several orders of magnitude lower than accelerator interface throughput that contemporary or upcoming interfaces such as PCIe, CXL, CCIX, or OpenCAPI (intend to) provide [3]. Therefore, the benefits of Apache Arrow may help alleviate bottlenecks in the context of FPGA accelerators as well.

A second advantage to using Arrow's standardized format exists. Because the in-memory format is derived from meta-data about the data sets, called *schemas*, we may also automatically derive a highly optimized hardware interfaces, (that could be considered as specialized DMA engines), from these schemas. From the perspective of an accelerator developer, these interfaces provide an easier starting point to interface with Arrow data sets, and in turn, to any of the languages supported by Arrow.

Access to objects/records and their fields can be expressed through tabular data set indices rather than the usual byte addresses, preventing the need to manually design units that perform tedious pointer arithmetic and perform the required requests on a memory interface. After supplying an index range of objects or records to process, the interface delivers streams of the exact data types expressed through the schema, rather than bus words. This allows the FPGA accelerator developer to fully focus on implementing the actual computational path of the accelerator only, rather than having to bother with the interface and the underlying infrastructure as well. This can normally be a cumbersome exercise, especially for data sets that consist of not just primitives such as *ints* or *floats*, but also contain more complex data types such as structure, lists and dictionaries (and any nested combination thereof).

Additionally, an advantage from building on top of the Apache Arrow ecosystem is that through Fletcher, high-performance FPGA accelerator integration is made available to all supported languages. Finally, a resulting advantage from delivering object or record fields as streams is that this integrates more naturally with dataflow oriented design styles, and potentially HLS-tools supporting this style, without having to write code that is interface specific.

We contribute the implementation of these ideas in the form of a fully open-sourced (including experiments, see [4]), vendor agnostic FPGA acceleration framework called *Fletcher*[1]. Fletcher is an FPGA accelerator framework specifically built on top of Apache Arrow, with the intent to not only solve challenges S1, S2, and S3 on the big data analytics framework integration side, but also to solve challenges H1, H2, and H3 on the hardware development side. This is illustrated in Figure 3.1. As such, Fletcher aims to decrease the integral time-to-solution for FPGA accelerated big data systems, as discussed in Chapter 1.

This chapter will describe the implementation of the Fletcher framework in detail. We will first give a high-level overview and discuss some related work in Section 3.2. We then describe some of the fundamental hardware components underlying the framework in Section 3.3. A large portion of the infrastructure is generated through software tools, that are described in Section 3.4. We give an example of how the software tools are used in Section 3.5. Specific applications and real use cases are described in Chapter 4.

3.2. HIGH-LEVEL OVERVIEW

A high-level overview of Fletcher, is seen in Figure 3.2. In this figure, the general compile-time and run-time flow is depicted, explained as follows:

DESIGN TIME OVERVIEW

At compile-time, a developer starts with an Arrow schema. From the schema, a template for the accelerator kernel implementation, and a complete infrastructure with interfaces to the kernel are generated. The interfaces provide high-performance, easy-to-use streams of requested data from the Arrow tables, and are generated by a tool called Fletchgen.

The term easy-to-use requires some elaboration. When accessing tabular data, one would prefer to do so through row indices rather than byte addresses. This has lead the Fletcher project to construct hardware components with configurable, streamable interfaces, that allow to provide a range of row indices, returning one or multiple streams of data corresponding to the types of Arrow tables. In contrast to a byte-addressable memory interface, this addresses challenge H2.

We briefly give an overview of the hardware structure, before diving into details in the next section. The Fletcher project contains some basic components that match

[1] A frequently asked question about the origin of the name may be answered by saying it needed to be something with an F for FPGA and it needed to have something to do with arrows.

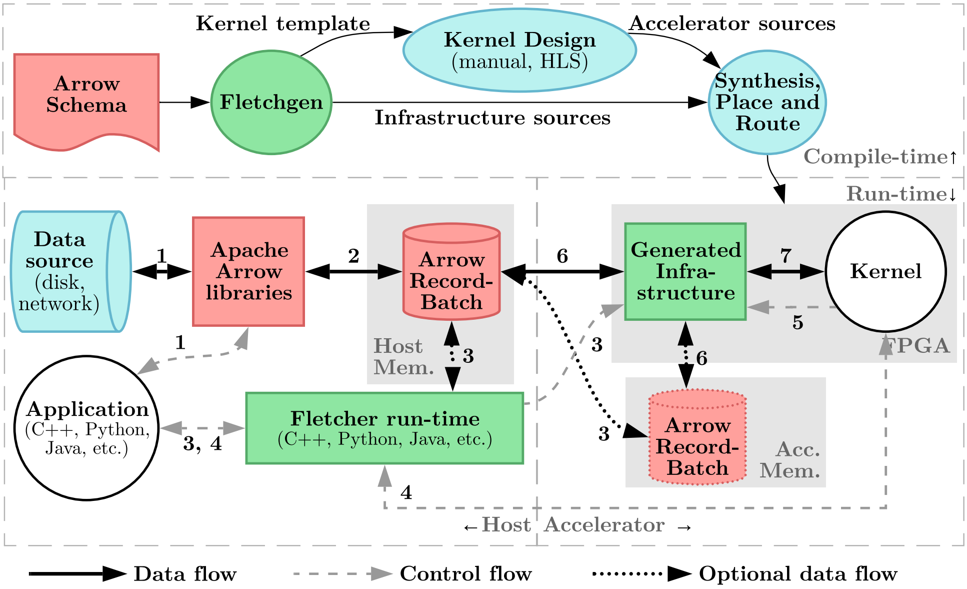


Figure 3.2: Architectural overview of Fletcher. Upper part of the figure shows the compile-time (development) flow, lower part of the figure shows the run-time flow for host system (left) and accelerator (right).

the abstractions of Arrow data sets. Based on an extensive vendor-agnostic library of streaming hardware primitives, Buffer Readers and Writers read/write from/to Arrow buffers. Combined according to the Arrow schema, they form Array Readers/Writers to read/write from/to Arrow arrays. This is more complicated than may seem initially due to the fact that the data types of the arrays may be variable-length and nested. It was still possible to capture the rather generative architecture in VHDL. Therefore, the configuration is passed to the Array Readers/Writers through a configuration string parameter, which is internally parsed and used to generate the appropriate structure corresponding to the Arrow array type.

The translation from Arrow types to the appropriate configuration string for Array Readers and Writers is done through a software tool called Fletchgen based on a hardware construction library called Cerata. Fletchgen also structurally combines Array Readers and Writers into RecordBatch Readers and Writers, to match the RecordBatch abstraction, It furthermore provides syntactically pleasing interfaces to the kernel, matching the field names of Arrow schemas, since VHDL does not allow to generatively change port names to something meaningful for a specific application. It then combines RecordBatch Readers and Writers into a full-fledged design. It finally generates the appropriate memory bus infrastructure and the control path that matches how the run-time library (explained next) automates most of the control flow.

Developers may add metadata to the Arrow schema and its fields to generate interfaces that, e.g. deliver multiple elements per cycle, or ignore schema fields altogether if they are not of interest. This allows the developer to make trade-offs between area, power and performance.

Fletchgen allows conversion of existing Arrow RecordBatches to a memory model for simulation that mimics a host interface and memory. In this way, a designer may perform hardware/software co-design of the kernel in simulation, agnostic of the final implementation platform.

To validate the correctness of the Array Readers themselves, schemas were generated randomly, where at each schema nesting level (within structs and lists) the complexity decreases on average such that eventually the nesting ends. Data sets based on this schema were generated randomly and random ranges of data are requested. The resulting stream outputs are checked with the expected outcome. Using this method, over ten thousand generated interfaces are validated in simulation.

RUN TIME OVERVIEW

At run-time, the enumerated steps in Figure 3.2 are taken:

1. Starting with a data source (e.g. a Parquet [5] file on disk), the data is loaded into memory.

2. Rather than loading the data set into a language native container (that would incur serialization overhead as soon as the data is needed in the accelerator), the application will ingest the data into memory formatted as an Arrow-based data set (e.g. a RecordBatch). Arrow library functions will place the data in host memory according to the schema and the format specification (if not already in the Arrow format). Note that this approach is in general required to exploit the benefits of Arrow as much as possible, irrespective of the use of Fletcher, and is in some big data analytics frameworks key to high performance [6].

3. The application can request the Fletcher run-time libraries to prepare the Arrow data set for processing on the accelerator. For some platforms this simply means passing virtual addresses of the buffers [7], and for other platforms this means a copy of the buffers must be made to accelerator on-board memory. This process is fully automated in the Fletcher run-time libraries. Basic use requires the user to only claim the platform / accelerator card, create a context in which the on-board memory is managed by the run-time, bind a host-side abstract representation of the kernel to a context, and provide the input RecordBatches as an argument to the kernel. Advanced users may use lower-level API calls to the Fletcher run-time system to e.g. place other data in the accelerator memory and control other data paths not generated through Fletcher.

4. The application can now issue commands to the kernel component of the accelerator. Commands include providing other types of arguments, reset, start, stop and poll for completion.

5. After the kernel receives the commands from the application, it can request a row or ranges of rows from the generated interface through a pipelined command stream.

6. The generated interface will request the desired data from the host memory or the accelerator on-board memory.

7. After receiving the data from the memory, the interface provides streams of data back to the kernel, containing the data from the requested rows and fields, in the form specified in the schema.

The last two steps can be reversed in case the kernel wants to write to an Arrow data set in memory.

3.2.1. RELATED WORK

Before we take a deep dive into Fletcher's internals in the next section, we will first consider some related work on FPGA acceleration frameworks. Several solutions to abstract away memory bus interfaces are commercially available and integrated into HLS tools (such as Xilinx' SDAccel and Intel's FPGA SDK for OpenCL). However, they have no inherent support for nested types that Arrow schemas can represent, and usually work well only with simple, C-like primitive types and arrays. HLS tools are also known to have problems dealing with dynamic data structures, as described in [8], that Arrow allows to express. At the same time, after Fletcher generates an interface that delivers streams which HLS tools can operate on very well.

Thus, while many commercial tools exist that automate infrastructure design, most of them are geared towards the HLS approach, but provide little help to users that for reasons mentioned above prefer to work with HDLs to describe their solution. Previous research has extensively investigated hardware interfaces for more generic, C-style dynamic data-structures through specialized DMA engines [9], but does not focus on integration with modern software frameworks from the big data analytics ecosystem analysis. To the best of our knowledge, Fletcher is the only open source FPGA accelerator framework that deals with challenge regarding the design to generate hardware infrastructure (defined as challenge H3 in Chapter 2) specifically in the context of big data analytics on tabular data sets for those that prefer an HDL design flow.

A number of frameworks do exist that help deal with the challenge related to platform portability (Chapter 2, challenge H1). We first give an overview of related work regarding this challenge, also shown in Table 3.1. This helps us compare Fletcher to existing frameworks, and stipulate the differences. We use the following criteria to include specific frameworks in our comparison:

- The framework is active and publicly available open-source.
- The framework targets datacenter-grade accelerator cards/platforms.
- The framework provides abstractions that provide some form of portability between such cards/platforms.

As shown in the table, there are currently a small number of other frameworks that adhere to these criteria. TaPaSCo [11] allows designers to easily set up systems that perform several hardware accelerated tasks in parallel. It is in some sense complementary to Fletcher, since (as will be discussed again later) Fletcher provides an AXI4 top-level for memory access, alongside an AXI4-lite for the control path of kernel, exactly fitting the integration style of TaPaSCo's processing elements. TaPaSCo furthermore allows design-space exploration to find optimal macroscopic configurations of the parallel kernels,

Framework	Focus	Targets	Ref.
Fletcher	HLL software integration, tabular data	AWS EC2 F1, OC-Accel, Xilinx Alveo	[10]
TaPaSCo	Parallel kernels, DSE	AWS EC2 F1, Various Xilinx-centric	[11]
Spatial	HDL (eDSL), DSE	AWS EC2 F1, Various Xilinx-centric, Intel Arria 10, other non-FPGA targets	[12]
OC-Accel	OpenPOWER/OpenCAPI systems	Alphadata 9V3, 9H3, 9H7	[13]

Table 3.1: Overview of open-source FPGA accelerator development frameworks

a feature that Fletcher does not have. It also allows to target a wide variety of (mainly embedded-oriented, but some datacenter-grade) FPGA accelerator cards, although currently only those that contain Xilinx FPGAs. Spatial [12] is mainly a domain-specific language embedded in Scala, tightly connected to the Chisel hardware description language [14]. The language provides a very high level of abstraction to design accelerators, and targets not only various FGPA accelerator platforms (of both Intel and Xilinx), but also CGRA-like and ASIC targets. Aside from not being a language itself, Fletcher differs from Spatial in the sense that it is less generic, and focuses only on abstractions to easily and efficiently access tabular data structures described in Arrow. OC-Accel [13], the successor of CAPI SNAP, does adhere to the criteria described, although it is still somewhat platform-specific, since it allows to target FPGA accelerator systems that have an OpenCAPI [15] enabled host system, typically found in contemporary POWER systems. OC-Accel is a target for Fletcher, aside from AWS EC2 F1 and Xilinx Alveo cards. We conclude the comparison by mentioning that Fletcher is a more domain-specific solution that only works for the tabular data structures of Apache Arrow. This prevents Fletcher from being used in other domains, although the lessons learned are of value when creating similar frameworks for other domains.

While Arrow is not the only framework following the trend of in-memory computation for big data frameworks (an overview can be found in [16]), it is a framework that is especially focused on providing efficient interoperability between different tools/languages. This allows the eleven languages supported by Arrow to quickly and efficiently transfer data to the FPGA accelator using Fletcher. State-of-the-art frameworks to integrate FPGA accelerators with structured data exist [17], although interface generation specific to the schema data types and serialization overhead are not discussed.

3.3. HARDWARE INTERNALS

Consider an accelerator to be the data sink in case an Arrow RecordBatch is being read. From the description in the previous section, we summarize a set of requirements for the generated interface:

1. **Row indexing:** The data sink is able to request table elements by using Arrow table

row indices as a reference. In turn, the data sink will receive the requested elements only.

2. **Streaming:** The elements will be received by the sink in an ordered stream.
3. **Throughput:** The interface can be configured to supply an arbitrary number of elements in a valid transfer.
4. **Bus interface:** The host-memory side of the interface can be connected to a bus interface of arbitrary power-of-two width.

The first requirement allows developers to work with row indices rather than having to perform the tedious work of figuring out the byte addresses of data (including potentially deeply nested schemas with multiple layers of offset buffers). Furthermore, it implies that elements are received in the actual binary form of their type, and not, e.g., as a few bytes in the middle of a host memory bus word (that are often 512 bits wide for contemporary systems). This allows the developer to not have to worry about reordering, serializing or parallelizing the data contained in one or multiple bus words.

The second requirement maps naturally to hardware designs that often involve data paths with streams of data flowing between functional units.

The third requirement allows multiple elements of a specific data type to arrive per clock cycle. For example, when a column contains elements of a small type (say a Boolean), it is likely the accelerator can process more than one element in parallel. This differs from Requirement 2 in the sense that the elements that will be delivered in parallel are part of the same request mentioned in Requirement 1. Furthermore, it can be that the top level element is a list of small primitive elements. Thus, one might want to absorb multiple of the nested elements within a clock cycle.

The last requirements allows the interface to be connected to different platforms that might have different memory bus widths. In the discussions of this work, we will generally assume that this width is set to 512 bits, since the platforms that Fletcher currently supports both provide memory bus interfaces of this size. However, Fletcher can also operate on wider or narrower bus interfaces.

VENDOR-AGNOSTIC HARDWARE LIBARY

Fletcher aims to be vendor-agnostic in order to thrive in an open-source setting. All designs are based on streaming interfaces, allowing for a natural dataflow-oriented style of design, commonly seen in FPGA accelerator systems. This requires custom streaming primitives that can perform the basic operations on streams. Commercial tools contain IP cores to support some (but not all) of these operations as well. However, to engage with an open-source oriented community, it is important to not force designs to use vendor-specific solutions. This causes the need for a custom streaming operations library that is maintained alongside Fletcher.

The most important streaming components are discussed in this subsection. The most basic primitives on which all other components are built, are as follows:

Slice A component to break up any combinatorial paths in a stream, typically using registers.

FIFO A component to buffer stream contents, typically using RAM.

Sync A component to synchronize between an arbitrary number of input and output streams.

The throughput requirement mentioned in the previous section dictates that streams must be able to deliver multiple elements per cycle (MEPC). To support this, and other operations, the previously mentioned primitives are extended by the set of following stream operators:

Barrel A pipelined component to barrel rotate or shift MEPC streams at the element level.

Reshaper A component that absorbs an arbitrary number of valid elements of an MEPC stream and outputs another arbitrary number of elements. This element is useful for serializing wide streams into narrow streams (or vice versa, parallelizing narrow streams into wide streams). The element can also be used to reduce elements per cycle in a single stream handshake or to increase (e.g. maximize) them. The implementation of the Reshaper uses the Barrel component.

Arbiter A component to arbitrate multiple streams onto a single stream.

Buffer An abstraction over a FIFO and a sync with a variable depth.

On top of the streaming components (especially the Arbiter and Buffer), a lightweight bus infrastructure has been developed to allow multiple masters to use the same memory interface. This bus infrastructure is similar to (and includes wrappers for) AXI-4, supporting independent read/write request and data channels and bursts.

Read/Write Arbiter Arbitrates multiple masters onto a single slave.

Read/Write Buffer Allows buffering of at least a full maximum sized burst to relieve the arbiter of any back-pressure.

3.3.1. COMPONENTS TO MATCH ARROW ABSTRACTIONS

IMPLEMENTATION ALTERNATIVES

Designing an interface to Arrow data could follow different approaches. A flexible approach would have a small customized soft processor generate the requests based on a schema or some bytecode that is compiled on the host. In this way, any schema (reasonably limited in size) could be requested, and schemas can be changed during run-time.

However, this approach would have several drawbacks. First of all, it would introduce more latency as it takes multiple instructions to calculate addresses and generate requests. Moreover, as developers can create schemas with fixed-width types of arbitrary length, allocating streams for the "widest" case is impractical. If one would supply the implementation with support for some very wide fixed-width type (effectively limiting the schemas that can be expressed already), it would cause a relatively large amount of area overhead for schemas with narrow primitives. For example, consider a hard-coded 1024-bit stream of which some schema only uses one bit. As schema data can be of many varieties, the streams would require run-time reordering of the elements coming from bus words. This involves relatively expensive parametrizations of the Stream Reshaper to support all possible cases of aligning arbitrary elements. Elements themselves must be restricted to be smaller than 1024 bits and only a fraction of RAM spent on FIFOs in the data paths is effectively used.

The aggregate of these drawbacks causes the proposed interface generation framework to completely configure the generated interface during compile-time. For this purpose, we introduce highly configurable components that correspond to abstractions seen in the Arrow software-language specific counterparts.

BUFFERS

Readers: As explained in Section 2.3, Arrow *buffers* hold C-like arrays of fixed-width data. We implement a component called a BufferReader (BR). The BR is a highly configurable component to support turning host memory bus burst requests and responses into fixed-width type MEPC streams. It performs the following functions:

- Based on the properties of the bus interface and the data type, perform the pointer arithmetic to locate elements of interest in the Arrow buffer.
- Perform all the bus requests desired to obtain a range of elements.
- Align received bus words.
- Reshape aligned words into MEPC streams with fixed-width data types.

An architectural overview of the proposed implementation of two BRs (in combination providing a setup to read variable-length types) is shown in Figure 3.3.

The top-level of a buffer reader contains the following interfaces, that are all pipelined streams:

Command (in) Used to request a range of items to be obtained from host memory by the BR. Also contains the Arrow buffer address and a special *tag*.

Unlock (out) Used to signal the completion of a command, handshaking back the command's original *tag*.

Bus read request (out) Used to request data from memory.

Bus read data (in) Used to receive data words from memory.

Data (out) A MEPC stream of data corresponding to an Arrow data type.

Reading from a values buffer, and reading from validity bitmap buffers (by instantiating a BR with element size one) is supported by the rightmost configuration of the BR as shown in Figure 3.3.

Here, a command stream is absorbed by two units: a bus request generation unit and an alignment and count controller. The bus request generator performs all pointer arithmetic and generates bus burst requests. The alignment and count controller calculates,

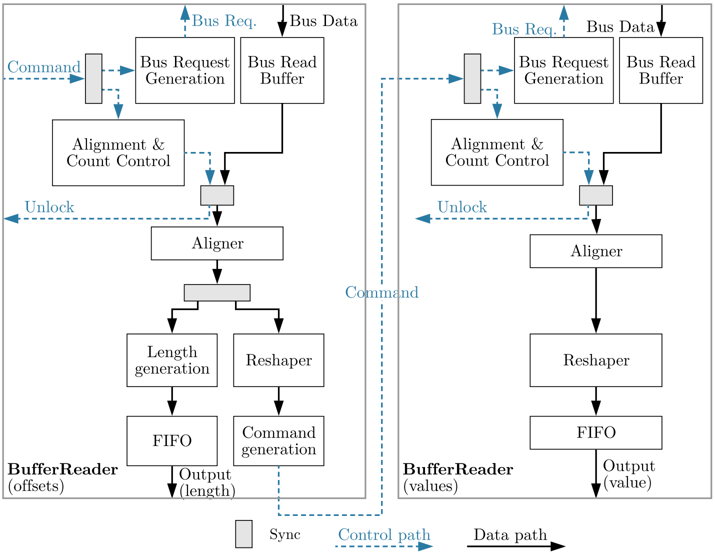


Figure 3.3: A BufferReader for an offsets buffer (left) and a values buffer (right)

based on the width of the bus and the type of elements, how much a bus word must be shifted (especially for the first bus word received), since some first index in the command stream might point to any element in a buffer. It also generates a count of valid items in the MEPC stream resulting from alignment. This is also useful when last bus words in a range contain less elements than requested.

Even though first and last bus words might not be aligned or do not contain all requested elements, after aligning and augmenting the stream with a count, the reshaper unit will shape a non-full MEPC stream into a full MEPC stream.

Furthermore, when the last bus word has been streamed to the aligner, an unlock stream handshake is generated to notify the accelerator that the command has been completed in terms of requests on the bus.

Offset buffers require the consumer of the data stream to turn an offset into a length. In this way, the consumer (typically the accelerator core logic) can know the size of a variable length item in a column. Therefore, for offset BRs, two consecutive offsets are subtracted to generate a length. Furthermore, BRs support the generation of an output command stream for a second BR. To generate this command stream, rather than generating a command for the child buffer for each variable length item, the BR requests both the last offset and the first offset in the range of the command first, before requesting all offsets in a large burst. The first and last offset can then be sent as a single command to the child BR, allowing it to request the data in the values buffer using large bursts.

Command (out) Used to generate commands for other buffers. This is useful when this BR reads from an Arrow offsets buffer.

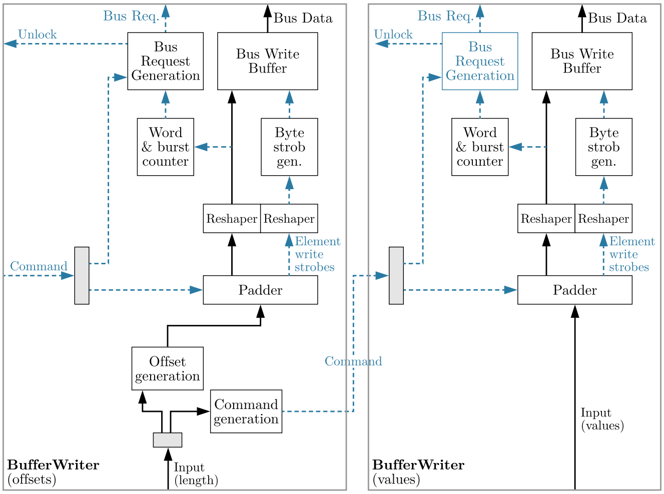


Figure 3.4: A BufferWriter for an offsets buffer (left) and a values buffer (right)

Writers: Complementary to BRs, we also implement BufferWriters (BW) that, given some index range can write to memory in the Arrow format. They contain the same interface streams as BR, except the data flow is inverted. An architectural overview of the proposed implementation of two BW is observed in Figure 3.4. Writing to a validity bitmap buffer or a values buffers requires the buffer writer to operate as follows (as seen on the right side of the figure).

When a command is given to the BW, the MEPC input stream is delivered to a unit that pre- and post-pads the stream to force the stream to be aligned with a minimum bus burst length parameter. Furthermore, it generates appropriate write strobes (only asserting strobes for valid elements). The elements and strobes are then reshaped to fit into a full bus word and sent to a bus write buffer. Note that sometimes it is unknown how long an input stream will be when the command is given. Therefore the command to the BW supports both no range or with range commands. At the same time this requires counting accepted bus words into the BusBuffer. A bus request generation unit uses this count to generate bus requests preferably when full bursts are ready, but if the input stream has ended, bus words are bursted out with minimum burst steps until the buffer is empty.

If the BW writes to an offsets buffer, it can be configured to generate offsets from a length input stream. This length input stream can optionally be used to generate commands for a child buffer. To achieve maximum throughput, the child command generation may be disabled, otherwise the child buffer writer will generate padding after the ending of every list in an Arrow Array containing variable length types.

ARRAYS

To support Arrow *arrays*, that combine multiple buffers to deliver any field type that may be found in an Arrow schema, we implement special components called Array Readers and Writers.

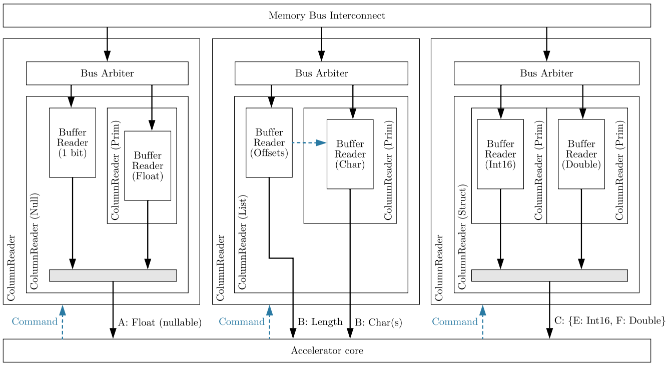


Figure 3.5: Resulting Array Reader configuration from the Schema in Figure 3.14

These Array Readers and Writers instantiate the BRs and BW resulting from a schema field. They furthermore support:

- Attaching command outputs of offsets buffers to values or validity bitmap buffers.
- Arbitration of multiple buffer bus masters onto a single slave.
- Synchronization of unlock streams of all buffers in use.
- Recursive instantiations of themselves. This, in turn, supports:
  - Nested types, such as `Lists<List<Type»`.
  - Adding an Arrow validity bit to the output stream.
  - Support Arrow *structs*, such as `Struct<List<Int16>, Float>`.

The Array Readers and Writers are supplied with a configuration string that conveys the same information as an Arrow schema. By parsing the configuration string, the components are recursively instantiated according to the top level type of the field in a schema. An example for the schema from Figure 3.14 is shown in Figure 3.5. Reading from the example RecordBatch (corresponding to the schema) will require three Array Readers. The manner in which they are recursively instantiated is shown in the figure. Here one can discern four types of Array Reader configurations:

Default A default Array Reader only instantiates a specific Array Reader of the top-level type of the corresponding schema field, but provides a bus arbiter to share the memory interface amongst all BRs that are instantiated in all child Array Readers.

Prim An Array Reader instantiating a BR for fixed-width (primitive) types.

Null Used to add a validity (non-null) bitmap buffer and synchronize with the output streams of a child Array Reader to append the validity bit.

List Used to add an offsets buffer that generates a length stream and provides a first and last index for the command stream of a child Array Reader.

Struct Used to instantiate multiple Array Readers, synchronizing their output streams to couple the delivery of separate fields inside a struct into a single stream.

Through the List and Struct type Array Readers, nested schemas may be supported. On the top level all streams that interface with the accelerator core are concatenated. A software tool named *Fletchgen* generates top levels for various platforms (including AWS EC2 F1 and OpenPOWER CAPI SNAP) that wraps around the Array Readers and Array Writers and splits the streams that are concatenated onto single signals vectors into something readable (using the same field names as defined in the schema) for the developer. A discussion of the inner workings of *Fletchgen* and the support for these platforms is outside the scope of this paper but the implementation may be found in the repository online [4]. The complement (in terms of data flow) of this structure is implemented for Array Writers. One additional challenge to Array Writers is that they require dynamically resizable Arrow Buffers in host memory, because it cannot always be assumed that the size of the resulting Arrow Buffers is known at the start of some input stream. This is an interesting challenge for future work.

Continuous integration All parts of Fletcher are open sourced. This allows all interested parties to submit changes to the hardware design. Part of improving the maintainability of the project includes bootstrapping of the build and test process in a continuous integration framework, where the simulator used is also an open-source project [18]. By using fully open-sourced tools in the collaborative development process, the threshold to get started with FPGA accelerators and Fletcher is lowered.

3.3.2. Results

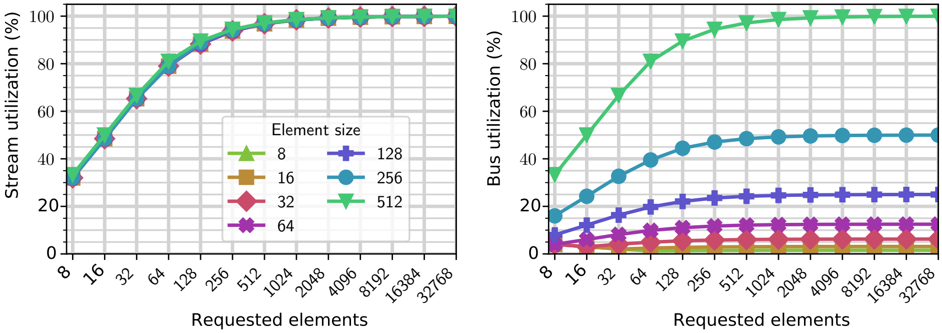
Functional validation

Because the number of schema field type combinations is virtually infinite (due to nesting), it is not trivial to validate the functionality of the framework. To obtain good coverage in simulation, a Python script is used to generate random schemas with supported types. The types decrease in complexity the deeper their nesting level, such that at some point the nesting ends with a primitive type. The resulting buffers are deduced from the schema, random content is generated and a host memory interface is mimicked. Random indices are requested from the simulated Array Readers, and their output streams are compared to the expected output. In this way, the correct functioning of over ten thousand different generated structures was validated.

Throughput

Array Readers/Writers for fixed-width types The main goal of the hardware components of Fletcher is to provide the output streams with the same bandwidth as the system bandwidth, if the accelerator core can consume it. In other words, the generated interfaces should not throttle the system bandwidth because of a sub-optimal design choice (like a sub-optimal in-memory format or a sub-optimal hardware component).

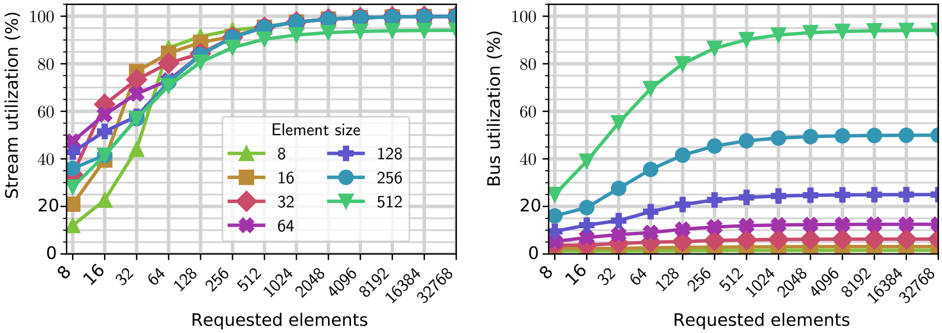
We simulate the throughput of Array Readers and Array Writers, assuming that we have a perfect bus interconnect, i.e. the bus delivers/accepts the requested bursts immediately and at every clock cycle a valid bus word can be produced. We measure the bus



(a) Near-optimal output stream utilization

(b) Memory bus pressure

Figure 3.6: Utilization for an Array Reader for various fixed-width types versus command range (each line represents a different fixed-width type).



(a) Near-optimal output stream utilization

(b) Memory bus pressure

Figure 3.7: Utilization for an Array Writer for various fixed-width types versus command range.

utilization and stream output utilization (in handshakes per cycle during the processing of a command) for different fixed-width types, as a function of the range of Arrow array entries requested through the command stream. We furthermore assume the accelerator core can handshake the Array Reader output or Array Writer input stream every cycle. The results of this simulation for a data bus width of 512 bits (as both platforms, AWS EC2 F1 and OpenPOWER CAPI SNAP, that Fletcher currently supports use this memory bus width) are shown in Figure 3.6b and 3.6a, where the bus utilization and output stream utilization is shown, respectively, for various fixed-width types. Similar measurements for the Array Writers are seen in Figure 3.7.

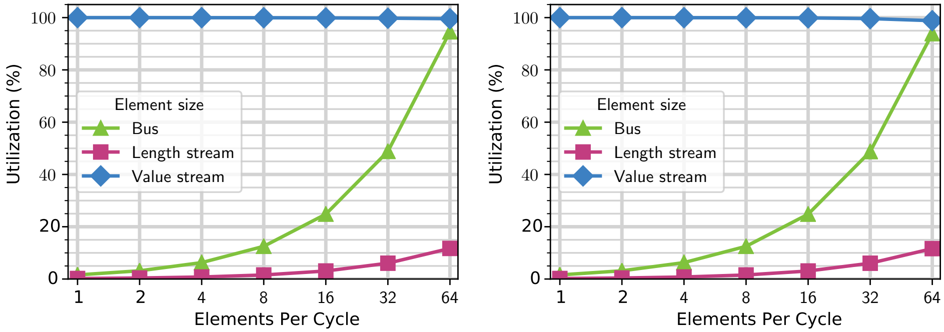
Initialization overhead and latency of both the Array Reader and Array Writer is present when the command only requests a short range of entries. However, once the range grows larger (a likely scenario in most big data use cases where massively parallel operators on data sets such as maps, reductions and filters are applied), the stream utilization becomes near optimal. As long as the element width is smaller than the bus width, maximum stream throughput is achieved, and as long as the element width is equal to the bus width, maximum bus bandwidth is achieved. We may conclude that an Array Reader for fixed-width types does not create a bottleneck if the accelerator core can absorb data at the system bandwidth rate. A developer using an Array Reader can now express access to an Arrow Array in terms of RecordBatch indices and will receive the exact data type as specified through the schema on the stream, without degradation of the system bandwidth.

Array Readers/Writers for variable-length types We simulate throughput of an Array Reader/Writer for an Arrow Array where the items in the Array are lists of primitive types. We choose the type to be a character (8 bits). We generate random lists between length 1 and 1024 and, in Figure 3.8, plot the utilization of the bus and the input/output streams as function of the elements-per-cycle parameter of this Array Reader/Writer. From these figures, we may observe that the value stream utilization is near-optimal, independent of the number of elements per cycle that it is configured for; as long as the memory bus can deliver the throughput, the accelerator core is fed at maximum throughput.

AREA UTILIZATION

For the same memory bus width as the supported platforms (512 bits), we synthesize Array Readers and Array Writers for various fixed-width types (W=8,16,...,512) and for various variable-length types (W=8 with EPC=64, W=16 with EPC=32, etc.) for a Xilinx XCVU9P device (that used in AWS EC2 F1 instances). The area utilization statistics are shown in Table 3.2.

The Array Readers/Writers require little area. Most configurations utilize less than one percent of the resources. Interestingly, Array Readers/Writers for small elements require more LUTs than wider elements on a wide bus. This is due to the reshaper and aligner units discussed in Section 3.3.1, requiring aligning and reshaping more MEPC stream element count combinations, increasing mux sizes. Designers may chose to reduce this number in the Array Readers and Writers themselves, but this requires an asymmetric connection to the memory bus interconnect, effectively moving the alignment functionality to the interconnect. Register usage increases when element size increases, since register slices



(a) Array Reader

(b) Array Writer

Figure 3.8: Bus and input/output stream utilization for an increasing elements-per-cycle parameter demonstrating utilization near 100%.

Table 3.2: Area utilization statistics for a Xilinx XCVU9P device

Type	Resource	W=8	W=16	W=32	W=64	W=128	W=256	W=512
Array Reader Prim(W)	CLB LUTs	0.30%	0.28%	0.26%	0.24%	0.22%	0.20%	0.21%
	CLB Registers	0.20%	0.20%	0.20%	0.20%	0.22%	0.24%	0.26%
	Block RAM (B36)	0.65%	0.65%	0.65%	0.65%	0.65%	0.65%	0.65%
	Block RAM (B18)	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%
Array Reader List of Prim(W)	CLB LUTs	2.34%	1.81%	1.46%	1.32%	1.03%	1.04%	0.78%
	CLB Registers	1.01%	1.01%	1.01%	1.01%	1.00%	1.00%	1.00%
	Block RAM (B36)	1.30%	1.30%	1.30%	1.30%	1.30%	1.30%	1.30%
	Block RAM (B18)	0.09%	0.09%	0.09%	0.09%	0.09%	0.09%	0.09%
Array Writer Prim(W)	CLB LUTs	0.20%	0.19%	0.19%	0.20%	0.20%	0.22%	0.23%
	CLB Registers	0.28%	0.28%	0.28%	0.28%	0.29%	0.31%	0.33%
	Block RAM (B36)	0.37%	0.37%	0.37%	0.37%	0.37%	0.37%	0.37%
	Block RAM (B18)	0.02%	0.02%	0.02%	0.02%	0.02%	0.02%	0.02%
Array Writer List of Prim(W)	CLB LUTs	1.03%	0.97%	0.91%	0.87%	0.80%	0.78%	0.52%
	CLB Registers	1.18%	1.12%	1.11%	1.11%	1.06%	1.06%	0.73%
	Block RAM (B36)	1.11%	1.11%	1.06%	1.06%	1.06%	1.06%	0.74%
	Block RAM (B18)	0.07%	0.05%	0.07%	0.07%	0.07%	0.07%	0.05%

on the path to the accelerator core match the width of the elements. Block RAM usage is the same for all configurations, because this depends on the maximum burst length that has been fixed to 32 beats for all configurations.

3.3.3. SUMMARY

The goal of the Fletcher framework is to ease integration of FPGA accelerators with data analytics frameworks. To this end, Fletcher uses the Apache Arrow in-memory format to leverage the advantages of the Arrow project, including no serialization overhead and interfaces to 11 different high-level languages. To support the wide variety of data set types that Arrow can represent, and to convert these data sets into hardware streams that are desirable by an FPGA developer, this work has presented a bottom-up view of a library of vendor-agnostic and open-source components. These components allow reading from tabular Arrow data set columns, by providing a range of table indices, rather than byte addresses, to refer to records stored in the tables. Fletcher is effective at generating these interfaces without compromising performance. It takes very little area to create an interface that provides an accelerator core with system bandwidth for any configuration of the Arrow data set. Fletcher significantly simplifies the process of effectively designing FPGA-based solutions for data analytics tools based on Arrow.

3.4. FLETCHER TOOLCHAIN

3.4.1. GENERIC FLETCHER HIGH-LEVEL ARCHITECTURE

We have so far described how Array Readers and Array Writers are generated, still using VHDL only. Although the components can be rather complex in nature already, they allow to merely access a single Arrow Array in a RecordBatch; one column in the tabular data structure. However, many applications require access to *multiple* columns, as well as *multiple* RecordBatches. Furthermore, accelerator kernels require a control path from host software as well.

With these requirements, a generic architecture of a Fletcher-based accelerator designs is presented in Figure 3.9. In this figure, the Array Readers/Writers (hereafter ArrayR/Ws) as described in the previous section are shown. We continue to explain the new components shown in the figure.

- **RecordBatch Reader/Writer**:
  RecordBatch Readers and Writers (hereafter RecordBatchR/Ws) are components that wrap around multiple ArrayR/Ws of a single RecordBatch. It may seem that the level of hierarchy that the RecordBatchR/Ws introduce does not necessarily have to exist, since the ArrayR/Ws can be operated independently of each other. However, a user may not want to issue a separate command to each ArrayR/W, but rather a single command to all ArrayR/Ws in a RecordBatch. The RecordBatchR/W allows to duplicate a single command stream into multiple command streams for each ArrayR/W, and allows to merge command responses into a single response stream as well. This ultimately adds support to access multiple columns.

- **Read/Write interconnect**:
  The Read/Write Interconnect components manage all memory interfaces coming

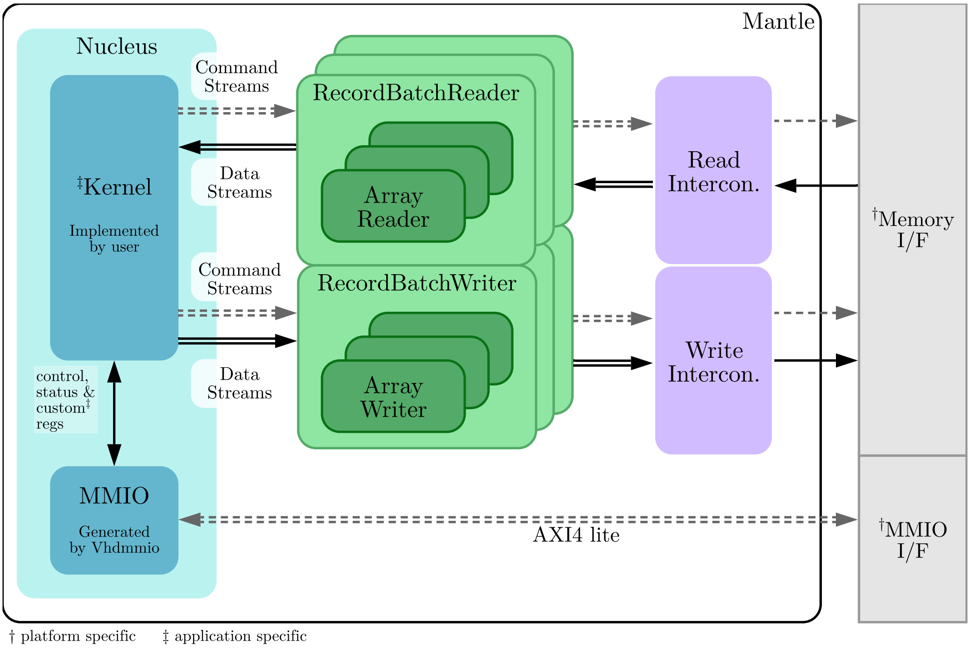


Figure 3.9: Generic top-level architecture of Fletcher accelerator designs

from the ArrayR/Ws. Since the memory interfaces of ArrayR/Ws may have various configurations, but the top-level memory interface typically only supports one configuration, serializers and parallelizers will automatically be inserted here. Furthermore, round-robin arbiters and buffers are instantiated in this component.

- **Nucleus**:
  The Nucleus component directly interfaces with the Arrow data streams, the command streams to the RecordBatchR/W, and with an AXI4-lite bus for memory-mapped I/O. Users may choose to implement their kernel at this level of abstraction, requiring them to insert their own MMIO controllers and fully manage the information on the command streams to the RecordBatchR/Ws themselves, including the addresses of the Arrow buffers in the memory. However, the philosophy of the Fletcher framework is to allow developers to express access to their data in terms of row indices, not having to worry about pointers (and pointer arithmetic). Therefore, by default, the Nucleus level abstracts the command streams of the RecordBatchR/Ws in such a way that the Arrow buffer addresses are hidden. To do so, it instantiates an MMIO controller that is used to pass information about buffer addresses from the host to the Nucleus. The MMIO controller is furthermore used to pass metadata about the specified RecordBatches and run-time information about the workload, such as the number of rows that a RecordBatch has, and a range of row indices for the kernel to operate on. Users may also pass or return application-specific information through these registers from/to the host machine.

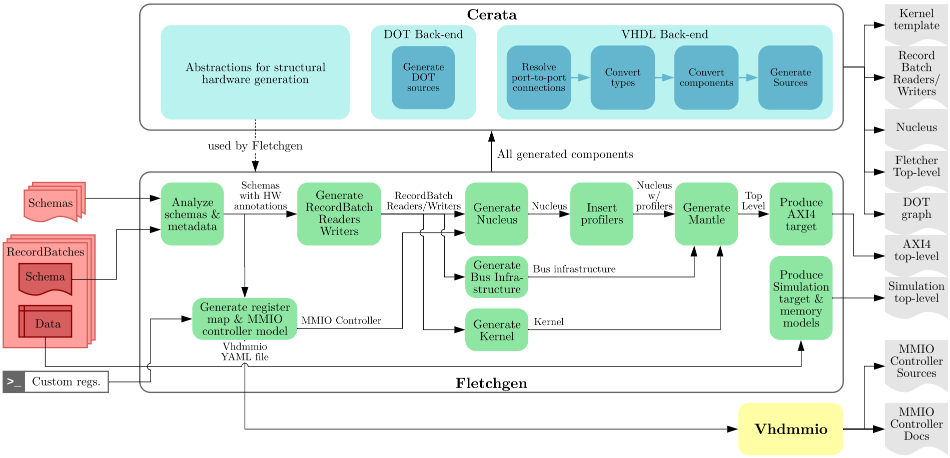


Figure 3.10: Overview of hardware generation components in the Fletcher tool-chain

- **Mantle**:
  The Mantle component wraps around all the other components, resulting in a top-level design that always has the same interface. In this way, supporting Fletcher on a new FPGA acceleration platforms is a matter of integrating the Mantle with the existing subsystems. Any generated Fletcher design can from that point onward be mapped onto that platform.

Through specialization of the previously described generic architecture shown in Figure 3.2, based on Arrow schemas, Fletcher faces challenge H3. However, to automate the specialization itself is a challenge on its own. Because of the large number of variations of designs that may be generated to accommodate multiple Arrow Arrays, multiple RecordBatches and the control path thereof, it is infeasible to implement a generic version of the design shown in Figure 3.2 in HDLs that vendor tools support.

To provide an agile and open-source hardware development experience to the users of Fletcher, a tool is required that is able to generate application-specific flavors of the generic architecture. It must furthermore be able to generate a platform-agnostic simulation environment, such that kernel implementations can be functionally verified independent of the target platform.

We therefore develop three new tools:

- Cerata; a generic hardware construction library providing high-level abstractions for structural hardware design.
- Vhdmmio: a generic MMIO controller generation tool taking a simple description of a register map, outputting VHDL sources with MMIO controller components that that can be connected to an AXI4-lite bus.
- Fletchgen: an Arrow-specific tool built on top of Cerata, using the abstractions provided to describe the generic architecture as shown in Figure 3.2, including the

RecordBatchR/Ws, the Nucleus, the interconnect infrastructure and the Mantle. It furthermore uses Vhdmmio for the control path from the host system through memory-mapped I/O.

These tools are part of the Fletcher hardware generation tool-chain, which we will continue to explain in more detail. A high-level overview of the tool-chain is shown in Figure 3.10.

3.4.2. CERATA

Cerata is an open source hardware construction library written in modern C++17. It is intended to be used only for structural hardware design, providing many abstractions for structural hardware generation. Structural designs can be described as a graph by connecting nodes representing ports, signals, parameters, literals and expressions. The graphs are hierarchical, such that they represent either components or instances. Cerata allows the expression of advanced interface types, supporting in particular nested streams that often emerge when converting nested Arrow data types into a form suitable for hardware. These can be connected with single lines of code, similar to how Chisel and SystemVerilog allow bulk connections.

Like Chisel that is hosted in Scala, Cerata allows already generated design to be inspected programmatically through its host language C++, resulting in what could be viewed as introspection. For example, it is possible to describe a component X, and during generation of another component Y that uses X, to inspect what ports X has in order to generate some structure that properly supports the instantiation of X. Note that this allows for a bottom-up generation approach, which is practically not possible in any traditional hardware description languages like VHDL[2]. There, all information has to be known at the top-level, trickling down to the lower levels of the hierarchy. The graph representations also allows for specific transformations to be implemented, one of which is to insert stream profilers, as we will discuss later. Since the rest of the Fletcher tool-chain is written in VHDL and C++ (the latter because the reference implementation of Arrow, that contains the latest features in general, is also written in C++), we have not used Chisel for this sort of introspective capability, since it outputs Verilog and is hosted in Scala.

After constructing and transforming graphs according to the needs of the user, Cerata can target two back-ends, a DOT [19] back-end to visualize the constructed graphs, and a VHDL back-end to generate structural VHDL.

3.4.3. FLETCHGEN

The input of Fletchgen are Arrow schemas and RecordBatches (that contain their schema plus data). All schemas are first checked for *required* metadata that is Fletcher-specific, and optional metadata. An overview of all *schema-level* metadata that Fletchgen understands is as follows:

- A schema name (required), used in the generation of HDL sources.

[2] Or is arguably incredibly esoteric in slightly more modern languages like SystemVerilog by writing low-level C support functions against the Verilog Procedural Interface.

- A schema access mode (required), specifying whether a user would like to read from a RecordBatch, or write to a RecordBatch.
- Memory interface specification (optional). This defines the properties of the bus infrastructure at the memory side of the interface. Properties include (amongst others) data width, address width and maximum burst length.

Additionally, schema *fields* can be annotated with the following metadata attributes:

- Ignore (true/false); a user not interested in a specific column of the RecordBatch can choose to ignore it. No hardware support or interface will be generated for this column. Note that this is an advantage of a columnar data storage system. Columns can be accessed completely independent of other columns.
- Elements per cycle; the maximum number of elements that can be handshaked in a single cycle on the output stream of this field.
- Length elements per cycle; the maximum number of list lengths that can be handshaked in a single cycle on the length stream of this field.
- Profile (true/false); a user may choose to insert stream profilers - units that gather statistics about the handshaking mechanism of (multi-element-per-cycle) streams, that can be translated into throughput. This helps users make performance/area trade-offs.
- Tag width; the number of bits used to identify commands and command responses.

After analysis of the metadata and after verification that specific properties (such as schema names) do not cause any conflicts, the hardware may be structurally described. Fletchgen generates the design from the bottom-up, starting with the instantiation of all the required ArrayR/Ws inside their corresponding RecordBatchR/Ws. In this step, the configuration string for the ArrayR/Ws is derived from the Arrow schema, using API calls provided by the Arrow library itself to traverse the tree of potentially nested field types.

The ArrayR/Ws are considered to be 'primitive' components as far as Fletchgen is concerned, i.e. they do not consist of other components that have to be generated (although their implementation is described with a very generative style of VHDL). ArrayR/Ws are described with VHDL, but this language does not allow port names to be generated. The data and control streaming interfaces therefore have nondescript names that are not easy to recognize for kernel developers.

The ports would preferably be named after the Arrow schema fields such that they are easy to recognize for kernel developers. Furthermore, because ArrowR/Ws can cause a variable number of streams to appear, these streams are concatenated onto port vectors for each type of stream, while it is more pleasing to get separate interface ports for every stream related to a specific Arrow field. We have therefore equipped Cerata with abstractions to concatenate multiple streams onto single ports and vice versa. These abstractions are used to eventually generate streaming RecordBatchR/W interfaces that have names corresponding to what Arrow field they were derived from, such that they become easily recognizable by users.

Also derived from the schema and its metadata is the memory-mapped I/O register map. Furthermore, users may supply additional arguments to reserve more custom registers in the register map through the command-line interface of the Fletchgen tool. All registers are 32-bits, controlled over an AXI4-lite interface from the host-side. Four categories of registers are mapped; default registers, schema-derived registers, custom registers, and profiling registers, as follows:

- **Default registers**; control, status, and two return value registers for results up to 64-bits wide. Since most of the target platforms have 64-bit addresses, this allows to pass a pointer to some resulting data structure, or a primitive return value. Resulting data structures laid out in the Arrow format would typically be passed to Fletchgen as a separate schema with access mode set to write.

- **Schema-derived registers**; the range of operation on each RecordBatch (first and last row index), followed by all Arrow buffer addresses, which we will call *RecordBatch metadata*. Because these registers are automatically set by the Fletcher run-time library, it is imperative that there is a unique order to the metadata, that is consistent between the hardware implementation and the software run-time. This is done by first sorting all schemas by name, and then stable sorting them by access mode. Since schema names must be unique for each access mode, the resulting unique ordering will make sure the hardware implementation corresponds to how the run-time library will set all metadata.

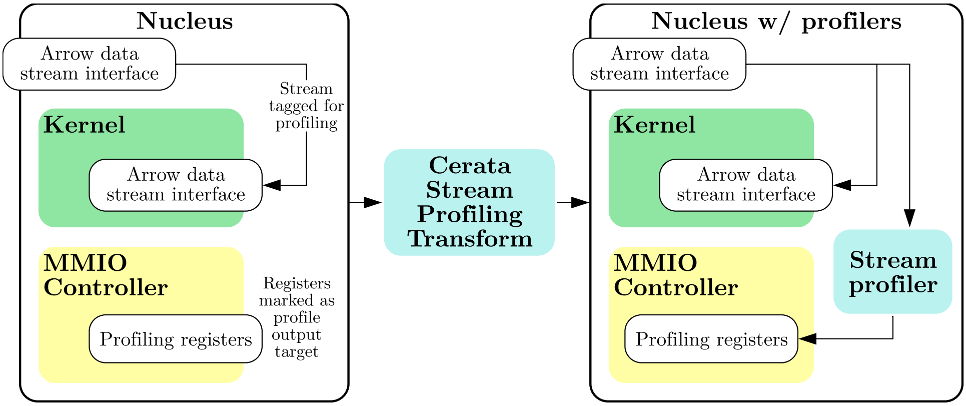
- **Custom registers**; the set of registers supplied by the user, for whatever purpose.

- **Profiling registers**; the registers that contain results of profiling Arrow data streams. These include a control register to start and stop profiling, as well as six measurement results; the number of elements transferred on the stream, the number of cycles the stream valid signal was asserted, the number of cycles the stream ready signal was asserted, the number of cycles both were asserted, the number of ’last’ signals handshaked (to count the number of stream packets transferred on variable-length types such as strings) and the number of cycles the profiler was enabled.

After the whole register map is known, Fletchgen generates a human-readable YAML-file that is passed to the Vhdmmio tool. This tool then generates an implementation of an MMIO controller according to the register map described above, and also outputs user-friendly documentation about the register map. Since the implementation of the MMIO controller is generated by this external tool, inside Fletchgen, it is considered to be a primitive component. Only a model of its interface is constructed, which is passed onto the next generation step.

The generated RecordBatchR/Ws and the MMIO controller now contain all information necessary to generate three more components. First, the memory bus infrastructure, that has to connect to the memory interface side of the RecordBatchR/Ws. Second, the Nucleus, that forwards the Arrow data and control streams to the third component; the Kernel. Note that the Kernel component is not implemented by Fletchgen, but must be implemented by the user.

The Nucleus is at first constructed without taking the stream profiler components into account, since inserting stream profilers is one of the *transformation* functions



Nucleus
Arrow data stream interface
Stream tagged for profiling
Kernel
Arrow data stream interface
MMIO Controller
Profiling registers
Registers marked as profile output target
Cerata Stream Profiling Transform
Nucleus w/ profilers
Arrow data stream interface
Kernel
Arrow data stream interface
Stream profiler
MMIO Controller
Profiling registers

Figure 3.11: Profiling transformation applied to the Nucleus

available in Cerata. As illustrated in Figure 3.11, after tagging streams with the profiling option, the profiling transformation function may be called by supplying the component implementation to transform. Optionally, references to signals where the stream profiler measurement outputs need to be connected can be given. When they are not given, the transformation will extend the component interface to contain the profiler measurement output signals. In the case of Fletchgen, we tag the streams between the Nucleus external interface and the Kernel corresponding to the field metadata supplied through the Arrow schema, and we supply the MMIO controller's profiling registers as output signals.

Now, a Nucleus instantiating the MMIO controller and the Kernel component with profiling registers is constructed. Together with the bus infrastructure that was generated, everything is tied together to form the Mantle — the Fletcher generic top-level.

Through the use of the Cerata hardware construction library, we have now implemented everything as shown in Figure 3.9, apart from the Kernel, which is left to the user. The design achieves the goal of the Fletcher framework; providing the user with hardware interfaces that correspond to the abstractions of Apache Arrow. They may now access RecordBatches through a command stream by only supplying row indices, and will read or write data over streams that correspond with Arrow's types.

3.4.4. RUN-TIME INTEGRATION

We continue describe how Fletcher is integrated during run-time, where challenge H1 related to portability must also be solved. An overview of the approach is shown in Figure 3.12, where show the example for two of the supported platforms, AWS EC2 F1 and OC-Accel. Because the top-level component, the Mantle, has the same interface for any Fletcher design, supporting multiple FPGA accelerator platforms is done creating platform-specific wrappers for the Mantle that are maintained in a separate open-source repository to prevent platform-specific code from contaminating the Fletcher code base. The platform specific low-level drivers to interact with the accelerator framework are abstracted, first through a low-level library in C, providing a common API for all platforms to the higher-level Fletcher platform-agnostic run-time libraries that are intended for

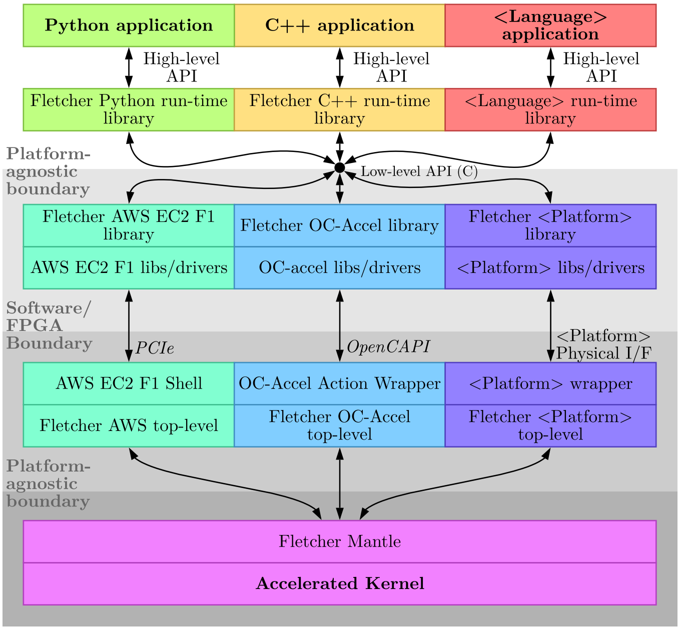


Figure 3.12: Platform-agnostic run-time stack

users. Fletcher run-time library dynamically searches and loads platform-specific versions of its low-level drivers, depending on what platform is available.

The currently supported languages include C++ and Python. The language-specific Fletcher libraries contain an API leaning heavily on Apache Arrow's abstractions. An example of how the accelerator is operated from Python is found in Figure 3.13.

During run-time, users only have to provide references to the RecordBatches of interest. The users only need to manually start the kernel and write and read values to their custom registers. These are placed into a queue, and automatically made available to the FPGA accelerator.

3.4.5. SIMULATION

To support users of Fletchgen with functional verification through simulation, note that in Figure 3.10, users may also supply Arrow RecordBatches. The Arrow schema that is contained within the RecordBatch will handled like any other schema, except the data in the RecordBatch will be used to produce a simulation top-level, wrapping the Mantle, and instantiating simulation-only memories that contain the RecordBatch data. The simulation top-level sets the buffer addresses and RecordBatch metadata automatically through the MMIO interface. It continues to send the start signal to the kernel, such that when the user is ready to debug the kernel, all data and control signals flowing in from the upper layers of the hierarchy are already handled. Only the custom registers are to be set appropriately by the user in the simulation top-level.

```python
import pyfletcher as pf

# Set up an auto-detected platform.
platform = pf.Platform()
platform.init()
# Create a Context for the data
context = pf.Context(platform)
# Queue input and output RecordBatches.
context.queue_record_batch(in_batch, out_batch)
# Enable the Context, causing data transfer
#   and MMIO registers to be set appropriately
context.enable()
# Set up an interface to the Kernel,
#   supplying the Context.
kernel = pf.Kernel(context)
# Start the kernel.
kernel.start()
# Wait for the kernel to finish.
kernel.wait_for_finish()
```

Figure 3.13: Example of using the Python run-time library to control the accelerator

3.5. USAGE EXAMPLES

To provide an example of the functionality described in the previous section, consider the following example application. Suppose we have two tables, where one table called `people` contains a unique key, names, ages and favorite food. The last item refers to a second table named `foods`, containing a unique key and food names. Suppose dinner must be cooked for all children based on their favorite food, we may query the tables for the names of all people under 12, and look up their favorite food.

When using the Fletcher framework to set up an accelerator implementation to solve this problem, we first have to define the Arrow schemas that describe the types of data each table will hold. An example of how this is done in Python is shown in Figure 3.14a. Note that one could use any language supported by Apache Arrow libraries to produce the schema, but for this article we choose Python because it is relatively succinct. The 30 lines of Python code hold enough information to produce Arrow schemas for our example. They can be passed to Fletchgen to generate a customized, application-specific version of the architecture presented in Figure 3.2.

On lines 3-15, the developer does not only define a Schema for the ’foods’ table, but also specifies some data it contains, and places the data inside an Arrow RecordBatch. As explained in the previous section, the data can be used to generate simulation models. The developer also supplies metadata on the `names` field, effectively tagging the resulting hardware streams to be profiled. Finally, the mandatory metadata are added; the access mode of the schema, in this case set to *read* from it, and the name of the schema to generate appropriate component and interface names.

On lines 16-24, the developer defines a schema, but since the question does not

```python
1  import pyarrow as pa
2
3  # RecordBatch describing the 'foods' input table:
4  foods = pa.RecordBatch.from_arrays(
5      # RecordBatch data for simulation:
6      [pa.array([10, 31, 32, 70], pa.uint16()),
7       pa.array(['apple', 'pear', 'banana', 'melon'])],
8      # RecordBatch schema:
9      schema=pa.schema([
10         pa.field('id', pa.uint16()),
11         pa.field('name', pa.string())
12           .with_metadata({'fletcher_profile': 'true'}),
13     ]).with_metadata({'fletcher_mode': 'read',
14                       'fletcher_name': 'foods'})
15 )
16 # Schema describing the 'people' input table:
17 people = pa.schema([
18     pa.field('id', pa.uint32())
19       .with_metadata({'fletcher_ignore': 'true'}),
20     pa.field('name', pa.string()),
21     pa.field('age', pa.uint8()),
22     pa.field('food_id', pa.uint16()),
23 ]).with_metadata({'fletcher_mode': 'read',
24                   'fletcher_name': 'people'})
25 # Schema describing the 'dinner' output table:
26 dinner = pa.schema([
27     pa.field('name', pa.string()),
28     pa.field('food', pa.string()),
29 ]).with_metadata({'fletcher_mode': 'write',
30                   'fletcher_name': 'dinner'})
```

(a) Schema definition example in Python

```vhdl
1  entity Kernel is
2    -- .. (generics omitted for brevity)
3    port (
4      -- The command stream to access the 'id'
5      -- column of the 'foods' table.
6      foods_id_cmd_valid    : out std_logic;
7      foods_id_cmd_ready    : in  std_logic;
8      foods_id_cmd_firstIdx : out std_logic_v..
9      foods_id_cmd_lastIdx  : out std_logic_v..
10     -- The incoming Arrow data stream for the
11     -- 'id' column of the 'foods' table.
12     foods_id_valid        : in  std_logic;
13     foods_id_ready        : out std_logic;
14     foods_id_last         : in  std_logic;
15     foods_id              : in  std_logic_v..
16     -- .. (other streams omitted for brevity)
17     -- Kernel control signals:
18     start, stop, reset    : in  std_logic;
19     idle, busy, done      : out std_logic;
20     -- Generic result signal going to the MMIO
21     -- controller:
22     result                : out std_logic_v..
23     -- RecordBatch metadata coming from the
24     -- MMIO controller:
25     foods_firstidx        : in  std_logic_v..
26     foods_lastidx         : in  std_logic_v..
27     -- .. (other metadata omitted for brevity)
28     -- Custom MMIO register input
29     age_threshold         : in  std_logic_v..
30   );
31 end entity;
```

(b) HDL output example in VHDL

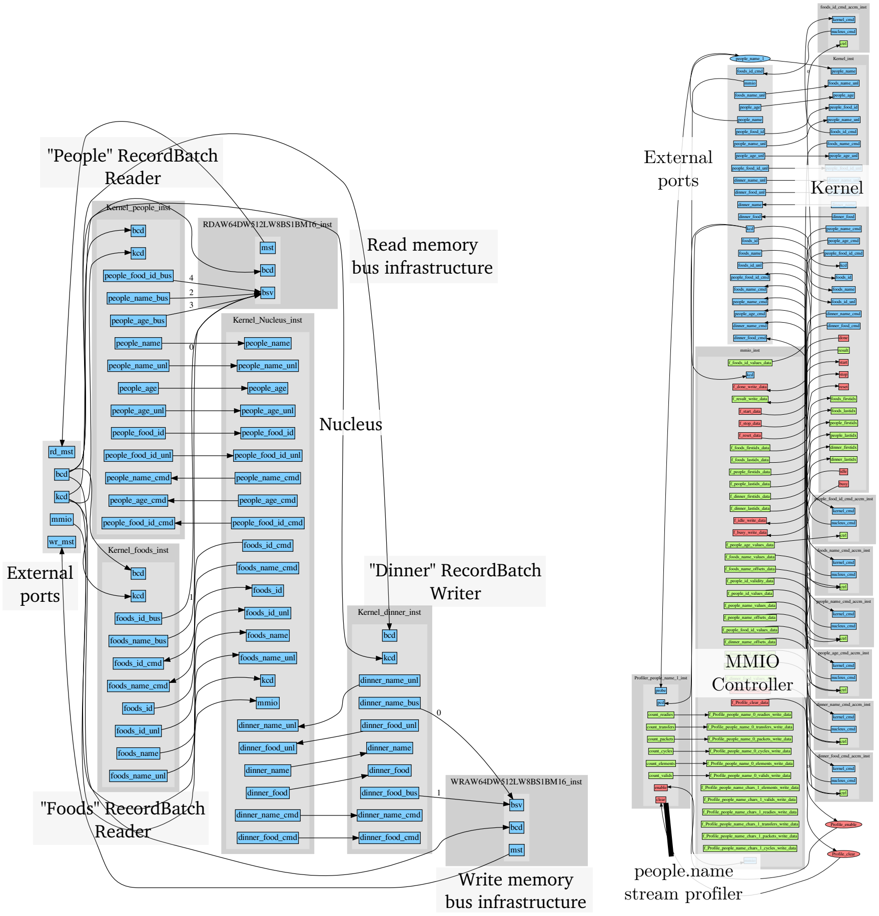
Figure 3.14: Examples of input and output of Fletchgen

involve returning the unique key of the people, merely their name, we have no use for this field. By supplying Fletcher-specific metadata, we may ignore this field, and no hardware will be generated to access this column.

Finally, on lines 25-30, the output schema is defined, with the access mode set to be able to *write* to the RecordBatch. For brevity, we have omitted 5 more lines involving saving the schema and RecordBatch to a file.

After providing Fletchgen with these schemas, we obtain many files that encompass the whole design as described in the previous section, corresponding to Figure 3.9, but specialized for the supplied schema. The only thing that the hardware developer has to do now, is implement the kernel, for which a template was generated. For our example, the template is shown in Figure 3.14b. Note that we have compacted the template for reasons of brevity, leaving out several rather detailed signal, and only show code related to the `foods` table's `id` field. The interfaces provided on the template allow the hardware developer to reason about the tabular data structures they are working with in terms of row indices, easing the development process.

In Figure 3.15a and 3.15b, we find the graphical representation of the design that was generated from the schemas in Figure 3.14a. This is the specialized version of the generic



(a) Mantle graph

(b) Nucleus graph

Figure 3.15: Graphical representation of example design in Cerata

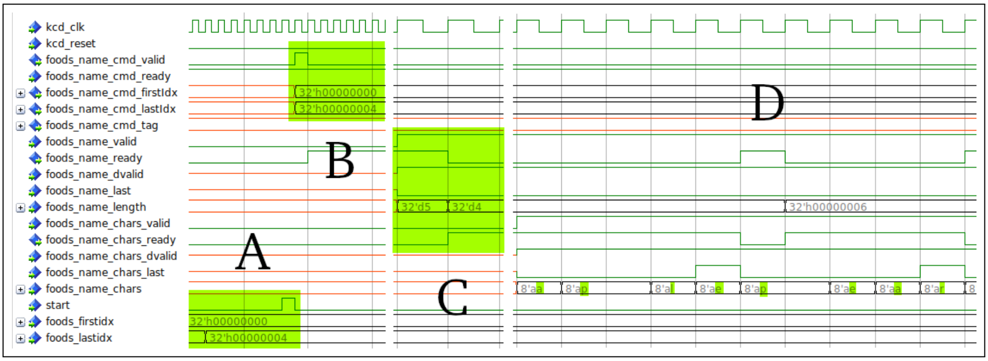


Figure 3.16: Streaming interface example, accessing the `foods name` field.

architecture presented in Figure 3.9.

To demonstrate the method of operation, suppose the kernel implementation requires all food `names` from the table. Fletchgen will generate a streaming interface appropriate for string data, using two streams; one for lengths, and another for the characters. We show the simulation waveforms of the access mechanism in Figure 3.16. Note that for brevity we have left out signals of the kernel component that are unrelated to the discussion, and have highlighted the main points of interest in the figures.

The RecordBatch metadata is automatically supplied through the MMIO controller before the kernel is given the start signal (A). The kernel can use the `foods_lasttidx` input to know the total size of the RecordBatch, to prevent reading out of bounds. However, if the developer wishes to parallelize the kernel, it is possible to also supply a `foods_firstidx`, such that each instance of this kernel can operate on its own part of the input tables and output tables. The kernel may send a command on the `foods_name_cmd` stream (B) to request the Arrow data, in this case all entries from the `name` column. Arrow data will start flowing into the kernel through the `foods_name` stream, that supplies string lengths (C), followed by the characters on `foods_name_chars` (D). Note that the first two food names appear on the character stream.

Starting off with interfaces that make sense w.r.t. the data structures the developer has to access contrasts heavily with the normal HDL-flow experience, where a developer typically starts off with a byte-addressable memory interface and a memory-mapped I/O interface. This demonstrates the Fletcher's ability to face challenge H2 as described in Chapter 2.

We finally demonstrate the ability to fine-tune the generated interface by making simple modifications to the Arrow schema. As we can see from Figure 3.16, the throughput of the character stream is relatively low, since only one character can be handshaked per cycle. Fortunately, the developer has annotated the field, as shown in Figure 3.14a, with the stream profiling option. After running simulation or the real implementation of this kernel, the developer may study the stream profile to find that the character stream provides a bottleneck to the whole system. In that case, the developer may simply annotate the Arrow field with the previously described option to provide multiple elements per handshake. Regenerating the design and making slight modifications to

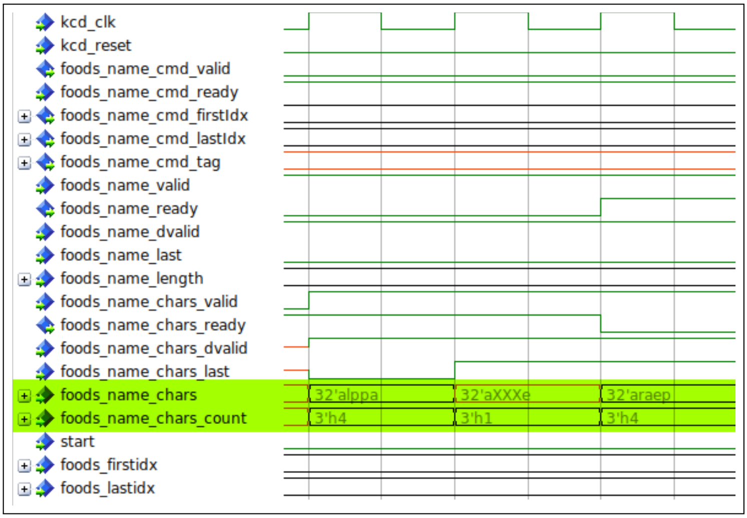


Figure 3.17: Accessing the `foods name` field with MEPH.

only the kernel will cause a `count` field to appear on the stream, as shown in Figure 3.17. The stream now allows to handshake four elements per transfer, with the count field indicating how many are valid. Note that the same amount of food names are handshaked as in Figure 3.16, although rather than taking ten cycles, they are now handshaked over three cycles, increasing the throughput of the stream at the cost of additional wires and control logic to support the wider interface.

On a final note, it is hard to properly quantify the reduction in development effort because of a large human dimension to such a measurement. To give a slight indication of the development effort saved, we could still look at the lines of code that were generated. From the thirty lines of Python seen in Figure 3.14a, Fletchgen and Vhdmmio generate 6304 lines of VHDL (not counting blank or comment lines), that are arguably human-readable and modifiable. This excludes the components that as far as Cerata is concerned are 'primitive'. The support library of hardware primitives and ArrayR/W's amounts to approximately 30K lines of code. To test a more extensive design, we have also captured all table schemas of the TPC-H benchmark suite in 120 lines of Python, resulting in Fletchgen to generate 33K lines of code necessary to provide access to all its tables.

3.6. CONCLUSION

In the previous chapter, we have discussed six specific challenges for FPGA accelerators to become widespread alternatives to existing computational solutions in the domain of big data analytics. In this chapter, we have discussed the Fletcher framework that aims to deal with these challenges with the help of Apache Arrow. Fletcher is built on top of Apache Arrow, providing a common, hardware-friendly in-memory format, allowing developers to communicate large tabular data sets between over eleven software languages without the need for copies, preventing (de)serialization overhead. Fletcher adds hardware accelerators to the list. Several low-level hardware components were designed to deal with the mentioned challenges for table columns, providing easy-to-use, high-performance interfaces to hardware accelerated kernels. The lower-level components are combined into a larger design, based on a generic architecture for FPGA accelerators that have tabular in- and outputs. Through an extensive infrastructure generation framework, specialized, data type-driven specializations of the generic architecture are generated, automating the tedious work of infrastructural design. The infrastructure generation tool made specifically for Arrow is built on top of a generic C++17 structural hardware construction library called Cerata, and on an MMIO controller generator framework called Vhdmmio. Developers can focus on the design of their kernels that are supplied with easy-to-use and high-performance hardware interfaces. The Fletcher tool-chain and run-time libraries drastically reduce the design and integration effort of FPGA accelerators into big data analytics pipelines while allowing the tabular data structures to be accessed at interface bandwidth. In that respect, Fletcher provides an integral approach to reduce the time-of-solution of FPGA accelerated system designs.

REFERENCES

[1] M. Maas, K. Asanović, and J. Kubiatowicz, *Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era,* in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17 (ACM, New York, NY, USA, 2017) pp. 138–143.

[2] The Apache Software Foundation, *Apache Arrow,* (2018).

[3] J. Peltenburg, A. Hesam, and Z. Al-Ars, *Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?* in *High Performance Computing,* edited by J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf (Springer International Publishing, Cham, 2017) pp. 220–236.

[4] Delft University of Technology, *Fletcher: A framework to integrate FPGA accelerators with Apache Arrow,* (2018).

[5] The Apache Software Foundation, *Apache Parquet,* (2018).

[6] Dremio, *Dremio - the missing link in modern data,* .

[7] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, *CAPI: A coherent accelerator processor interface,* IBM Journal of Research and Development **59**, 7 (2015).

[8] F. Winterstein, S. Bayliss, and G. A. Constantinides, *High-level synthesis of dynamic data structures: A case study using Vivado HLS,* in *2013 International Conference on Field-Programmable Technology (FPT)* (2013) pp. 362–365.

[9] G. Weisz and J. C. Hoe, *CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing,* in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)* (2015) pp. 1–8.

[10] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee, *Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow,* in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (2019) pp. 270–277.

[11] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, *The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems,* in *Applied Reconfigurable Computing,* edited by C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz (Springer International Publishing, Cham, 2019) pp. 214–229.

[12] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, *Spatial: A language and compiler for application accelerators,* in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018 (Association for Computing Machinery, New York, NY, USA, 2018) p. 296–311.

[13] A. Castellane and B. Mesnet, *Enabling fast and highly effective fpga design process using the capi snap framework,* in *High Performance Computing,* edited by M. Weiland, G. Juckeland, S. Alam, and H. Jagode (Springer International Publishing, Cham, 2019) pp. 317–329.

[14] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, *Chisel: Constructing hardware in a scala embedded language,* in *DAC Design Automation Conference 2012* (2012) pp. 1212–1221.

[15] OpenCAPI Consortium, *OpenCAPI technical specifications,* (2018).

[16] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, *In-memory big data management and processing: A survey,* IEEE Transactions on Knowledge and Data Engineering **27**, 1920 (2015).

[17] M. Owaida, D. Sidler, K. Kara, and G. Alonso, *Centaur: A framework for hybrid cpu-fpga databases,* in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2017) pp. 211–218.

[18] T. Gingold, *Ghdl vhdl 2008/93/87 simulator,* (2018).

[19] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, *Graphviz— open source graph drawing tools,* in *Graph Drawing,* edited by P. Mutzel, M. Jünger, and S. Leipert (Springer Berlin Heidelberg, Berlin, Heidelberg, 2002) pp. 483–484.

[20] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, *Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow,* in *Applied Reconfigurable Computing,* edited by C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz (Springer International Publishing, Cham, 2019) pp. 32–47.

[21] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, *Generating high-performance fpga accelerator designs for big data analytics with fletcher and apache arrow (under review),* Journal of Signal Processing Systems (2020).

[22] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, and et al., *The FitOptiVis ECSEL Project: Highly Efficient Distributed Embedded Image/Video Processing in Cyber-Physical Systems,* in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19 (Association for Computing Machinery, New York, NY, USA, 2019) p. 333–338.

PERSONAL CONTRIBUTIONS, ACKNOWLEDGEMENTS AND FUNDING.

This chapter is based on two published articles [20][10] and one under review at the time of writing [21]. J. van Straten made significant contributions to most hardware designs and implementations presented in this chapter, and contributed the *vhdmmio* and *vhdeps* tools. M. Brobbel contributed to the development operations around the Fletcher framework. L. Wijtemans helped debugging Fletcher, especially on the AWS EC2 F1 system. L. van Leeuwen helped debugging Fletcher, and implemented the Python run-time library. J.W. Peltenburg wrote all papers, made significant contributions to most hardware designs, wrote Fletchgen, Cerata and all other platform libraries and run time libraries.

The authors would like to thank IBM, Xilinx and AlphaData for their support.

This work was partially supported by the European Commission in the context of the ARTEMIS project ALMARVI (project #621439). This work was partially funded by Xilinx. This work is part of the FitOptiVis project [22], funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162.

4

Applications

In this chapter, we demonstrate various FPGA-accelerated applications developed with the help of (components of) Fletcher. We first discuss an accelerator implementation for regular expression matching, where we especially focus on the serialization overhead associated with data exchange between host system software running in C++, Python and Java. The use of Apache Arrow and Fletcher to prevent this overhead results in an increased accelerated application throughput of between 1.3× and 49×, compared to running the accelerated application with serialization from a traditional in-memory layout. Second, we implement an accelerator for k-means clustering, showing a benefit of up to 2.7 ×, even though this application is more computationally intensive, typically less impacted by serialization overhead. Third, we demonstrate an accelerator writing strings from FPGA into host memory. The design demonstrates the ability to scale up Fletcher data streams to match system bandwidth, where we achieve a throughput of 12 GB/s. Fourth, we briefly explore a simple HLS kernel attached to a Fletcher streaming interface. Fifth, we extensively study the PairHMM algorithm that is applied in genomics, and accelerate the algorithm with an FPGA accelerator, later using Fletcher to provide easy to use interfaces to it, and modifying the arithmetic units to make use of the new posit floating-point format, increasing numerical precision. This application demonstrates the specialization advantage of FPGA accelerators achievable with a much better time-to-solution than would be possible when designing an ASIC solution. The design provides a fifteen-thousand times speedup over a CPU that is required to perform the arithmetic with software emulation. Finally, we explore an FGPA-accelerated application for Apache Parquet to Apache Arrow conversion using Fletcher. Since the bandwidth of storage systems is also rapidly increasing, we find that CPUs can no longer keep up with the storage interface bandwidth, and we off-load the conversion operation to an FPGA, that may be directly attached to storage in the future. This results in a file decoding throughput of 6 GB/s for the FPGA accelerator compared to 2 GB/s for an optimized CPU implementation.

4.1. INTRODUCTION

In this chapter, we implement six applications with the help of (parts of) the features of Fletcher described in the previous chapter. The first three applications, regular expression matching, k-means clustering, and writing strings to memory at high bandwidth, specifically focus on the increased performance when serialization overhead is prevented through the use of Arrow. This is described in Sections 4.2, 4.3, and 4.4, respectively. We briefly explore integrating with a commercial HLS tool in Section 4.5.

The remainder of the chapter focuses more on applications themselves, where Fletcher was used to decrease the design effort. We continue to describe the acceleration of a larger application in genomics, called Variant Calling, in Section 4.6, that was later extended to make use of Fletcher for easy integration, and posit arithmetic for increased arithmetic precision. Up to Section 4.6.7, an in-depth application-specific discussion and kernel design follow, somewhat orthogonal to the contributions related to Fletcher. From that section onward, we introduce the Arrow schema and upgraded architecture to make use of Fletcher and posit arithmetic.

Finally, in Section 4.7, we explore how a widely used file format in big data analytics, called Apache Parquet, may be decoded in an FPGA and, through the use of components from Fletcher, immediately written to host memory in the Apache Arrow format. The contributions in that Section have been explored based on the increasing amount of I/O bandwidth observed for non-volatile storage systems.

It must be mentioned that these applications were developed alongside the many features of Fletcher described in Chapter 3, starting from the moment the Array Readers were available. Therefore, not all previously presented features were used in each of the presented accelerator designs.

We have used a variety of datacenter-grade systems, interfaces, and FPGA accelerator cards to perform the experiments, of which an overview is shown in Table 4.1.

POWER8/ADM7V3	
Node	IBM Power System S824L (8247-42L)
Interface	CAPI1.0 over PCIe3 x8
Card	AlphaData ADM-PCIE-7V3
FPGA	Xilinx XC7VX690
Amazon EC2 F1	
Node	EC2 F1 instance, Intel Xeon E5-2686 v4 CPU
Interface	PCIe3
Card	undisclosed
FPGA	Xilinx XCVU9P
POWER9/ADM9V3	
Node	IBM POWER9 "Barreleye"
Interface	CAPI2.0 over PCIe 3 x16
Card	AlphaData ADM-9V3
FPGA	Xilinx VU3P
POWER9/ADM9H7	
Node	Inspur FP5290G2, 2× POWER9 Lagrange 22-core CPU
Interface	OpenCAPI
Card	AlphaData ADM-PCIE-9H7
FPGA	Xilinx XCVU37P

Table 4.1: Overview of the systems used in this chapter

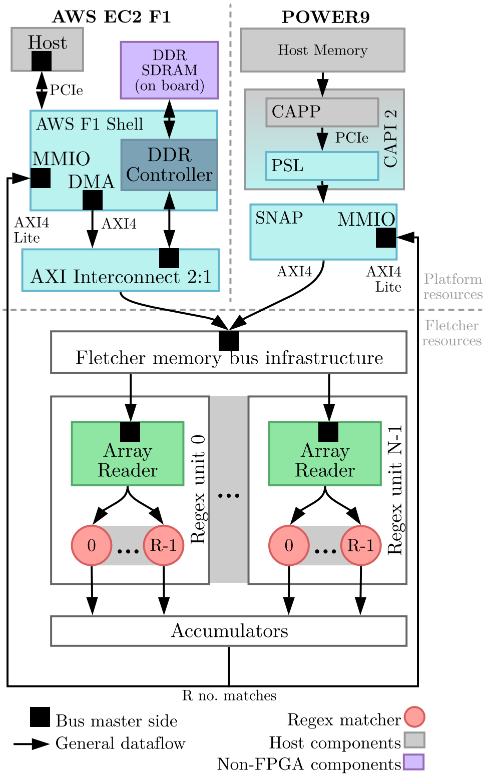


Figure 4.1: Architecture of the regular expression matching experiment. For AWS EC2 F1: N=16, POWER9+CAPI2.0: N=8.

4.2. REGULAR EXPRESSION MATCHING

Consider a use case where the number of matches to a regular expression in a large collection of strings is of interest. For example, one wants to know what the most talked-about house pet on a social network is. To do this it is required to match a large collection of strings to some regular expressions of the form `.*(?i)kitten.*`, `.*(?i)puppy.*`, etc., and count the number of matches (`"?i"` stands for any case of the characters in the expression). In this first application, a large collection of (tweet-sized) strings is matched to a set of sixteen regular expressions.

The number of matches are counted for each regular expression. It is an application that is fully streamable and generally performs extremely well on an FPGA — hence any serialization overhead can penalize its potential performance tremendously. With this example application, we can measure the performance of Array Readers that fetch variable-length objects (UTF-8 strings). The software kernels use the fastest regex matching libraries we could find. In C++ we use the RE2 library [1] and spread the workload over all available CPU threads. We use the Python wrappers for the RE2 library as well and the standard multiprocessing module to spread the workload over all hardware threads.

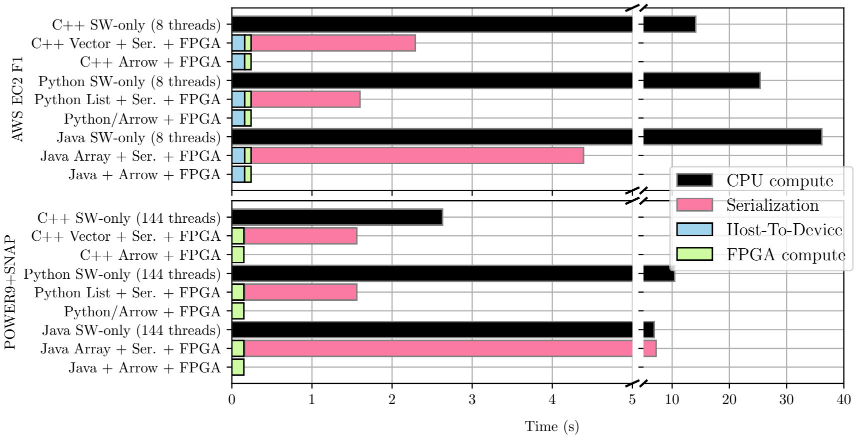


Figure 4.2: Run-time components of regular expression matching accelerator

In Java the built-in regex matcher is fastest, and has also been parallelized over all CPU threads.

In the FPGA implementation, for which the architecture is shown in Figure 4.1, we place multiple streaming regex matching units in parallel where each unit has an Array Reader configured to deliver four characters per cycle at 250 MHz. This setup matches the peak theoretical throughput of 16 GB/s for the on-board DDR interface. A data set with random length strings between 0-255 with a total size of 1 GiB is used as an input. Both platforms are shown in Figure 4.1, where the upper part of the figure is divided and specific to the corresponding platform, while the lower part is equal for both setups. Herein one may note the advantage of Fletcher providing a platform-agnostic environment, with respect to challenge H1.

VHDL implementations of the regular expression (regex) matchers are generated by a tool that uses a non-deterministic finite automaton approach, which can be found online [2]. All regex units operate their Array Readers in parallel to supply data streams that are duplicated to perform sixteen different regexes in each unit. Thus we exploit parallelism at the column level by working in different parts of the column at the same time, and at the matching level by attempting multiple matches on the same data. For the AWS EC2 F1 system, 16 Regex units are deployed (256 matchers in parallel), while for the POWER9+CAPI system, 8 regex units are deployed (128 matchers in parallel).

From the run-time measurements, shown in Figure 4.2 and throughput measurements, shown in Table 4.2, we find that the FPGA kernel vastly outperforms the CPU implementation, as expected. However, to get the data set to the accelerator, in the traditional case, *we must first serialize it.* The serialization throughput for each language is below 1 GB/s, while the EC2 F1 platform has a copy bandwidth of over 7 GB/s. Once the data is on the on-board memory, the parallel Array Readers are able to stream the data to the regex units at over 14 GB/s (achieving almost 90% of the peak bandwidth). Through the use of the Arrow in-memory format and interfacing with the data through the use of Fletcher, the end-to-end speedup improves by over 9×, up to 18×, depending on the

		Throughput (GB/s)				Speedup		
System	Language	Native data set w/ CPU	Serialization	FPGA Copy	FPGA Kernel	w/ Serialization	Arrow/Fletcher	Improvement
AWS/F1 (16 regex units)	C++	0.08	0.55	7.13	14.27	6.18	59.73	9.67
	Python	0.04	0.83	7.17	14.28	15.93	107.73	6.76
	Java	0.03	0.27	7.13	14.27	8.24	152.91	18.56
P9/SNAP (8 regex units)	C++	0.43	0.81	n/a	7.61	1.70	17.78	10.44
	Python	0.11	0.81	n/a	7.61	6.77	70.72	10.45
	Java	0.16	0.16	n/a	7.61	0.95	46.49	48.69

Table 4.2: Regular expression matching results

software run-time system. A similar advantage may be observed in the POWER9/SNAP case. In the particular case of the Java implementation on this platform, serialization dominates so much, that even though the accelerator exhibits an almost two orders of magnitude higher throughput, it would not be worthwhile to use the accelerator when serialization has to take place. This is mainly due to a very low serialization throughput, and as a result, using Fletcher yields a very high improvement factor.

Additionally, we find the FPGA resource utilization of each Array Reader for this example to be 1.45% CLBs and 0.21% BRAM tiles for the XCVU9P). Further details on resource utilization of a wide variety of ArrayReader/Writer configurations are be found in [3]. In summary, the CLB utilization ranges from 0.02% for primitive types with the width of the data bus to 2.34% for Array Reader for lists of primitives able to deliver 64 elements per cycle.

4.3. K-MEANS CLUSTERING

We perform K-means clustering (only on AWS/F1) of a data set of integer feature vectors; a common kernel in data analytics that is computationally intensive. The algorithm is of a more iterative nature; it is not fully streamable and therefore the impact of serialization is expected to be less dominant. At the same time, using Fletcher we may generate an easy-to-use interface that delivers streams of vectors of which the lengths is defined during run-time.

The C++ implementation uses a vector of feature vectors as input data and performs the clustering using a parallelized implementation on all hardware threads. The Python implementation wraps the C++ implementation through Cython. The Java implementation uses an `ArrayList` of `ArrayLists` as an input dataset and is also multithreaded. The FPGA implementation processes one feature vector per clock cycle, where up to 16 features can be processed in parallel from the input stream received from the Array Reader. For every iteration of the K-means algorithm, the whole data set is requested through the Array Readers. For our dataset of aprox. 1 GiB of feature vectors, the number of iterations was 25, and thus we may calculate the average bandwidth per iteration for all implementations.

Table 4.3: K-means clustering results

	Avg. GB/s		Total run-time (s)		
Language	CPU	FPGA	CPU	FPGA (w/ ser)	FPGA (w/o ser)
C++	1.40	11.15	19.24	6.08	2.55
Python	1.29	11.15	20.77	8.07	3.03
Java	1.00	11.15	26.92	3.88	2.55

The results of this measurement are shown in Table 4.3. It can be seen that the bandwidth of the Array Reader grows close to the peak bandwidth (delivering up to 70%, although computational aspects of the implementation are also included in this measurement). The results show that even for a computational intensive algorithm like K-means, the benefit can be substantial (up to 2.7× in this particular case).

4.4. STRING WRITER

In this example, we consider *writing* to Arrow RecordBatches from FPGA. Use cases include the FPGA being the data source or being in the data-path from another source to host memory (e.g. data coming from a network interface or storage). The data source contains a set of string lengths and a set of string characters (similar to e.g. how uncompressed Parquet files store strings). Our intent is to measure how fast ColumnWriters can write variable-length objects into a format that is usable by the software-language run-times that Arrow supports.

Because connecting the FPGA to an actual flash drive or network interface is outside the scope of this work, we mimic such an input in FPGA by generating a character stream with 64 characters per cycle (at 250 MHz) and another stream with pseudo-random lengths between 0-255, resulting in a total data size of approximately 1 GiB. The length stream is generated uniformly random between 0-255. This results in the 64-character input stream where every handshake on average only has 75% valid input data, resulting in a peak input rate of 12 GB/s. In software, the time to deserialize the same data source to a language native container (C++: `vector<string>`, Python: `list` of strings (using Cython), Java: `Array<String>`, all pre-allocated where applicable) as well as to an Arrow RecordBatch is measured (in the Python case by wrapping the C++ implementation).

From the measurement shown in Table 4.4, it can be concluded that the Arrow format itself already gives a performance benefit because it does not require the need to allocate memory for each string object separately. The Array Writers of the FPGA implementation are able to generate the Arrow RecordBatch at an even higher throughput of almost 10 GB/s, slightly over 80% of the average input bandwidth of 12 GB/s. The device-to-host bandwidth of the AWS/F1 system only delivers 2.53 GB/s at the time of performing the experiment, causing a bottleneck for the FPGA implementation. This is expected to be increased, while the AWS/F1 system is further developed. For the P9/SNAP system, a more modest speedup of 1.3× is observed.

Table 4.4: String writer results

System	Language	Throughput (GB/s)			
		To native container	To Arrow RecordBatch	FPGA copy	Total (Arrow RecordBatch)
AWS/F1	C++	0.85	2.53	-	2.53
	Python	0.96	2.60	-	2.60
	Java	0.59	1.81	-	1.81
	FPGA	-	9.76	2.75	2.15
P9/SNAP	C++	0.76	7.52	-	7.52
	Python	1.60	7.68	-	7.68
	Java	0.28	3.96	-	3.96
	FPGA	-	9.76	-	9.76

4.5. HLS-BASED FILTER

We have explored augmenting an existing commercial HLS tool (Vivado HLS) with Fletcher. An Arrow RecordBatch was created with two columns containing a string and a third column containing an integer. On this RecordBatch, an SQL-like query is be performed that exactly matches the contents of one string and the integer, and returns the other string column.

Initially, the kernel is described as a (HLS-oriented) C++ function that has pointer arguments to the used Arrow buffers. The HLS tool initially cannot compile this kernel as there is no static information about the size of the buffers. After adding a pragma for each of the buffer pointers we are able to compile an implementation that communicates with a memory bus. The code for this variant of the filter kernel is shown in Figure 4.3. Assuming an off-chip memory latency in the order of a hundred nanoseconds (∼25 cycles at 250MHz), this kernel incurs memory latency for the outer loop that iterates over all strings. This results in an outer loop iteration latency of at least 49 cycles with an inner loop iteration latency of two cycles. Only a fraction of the cycles are spent on actual work; the kernel is memory latency bound. Additional pragmas and rewriting the kernel in a specific way would allow to optimize this behavior, even as so far to possibly write additional functions that mimic Fletcher's approach. However, Fletcher helps automate this process and overcomes the need for rewriting the kernel.

In a second implementation, using *Fletchgen*, an interface is automatically generated based on an Arrow schema. The interface provides the ability to write the kernel as a C++ function with `hls::stream<type>` arguments, as shown in Figure 4.4. The input streams provide the properties of the string; a length and character stream for each string, and a stream with the integer. After the filter step has been performed, the kernel may push characters and lengths into the output stream. Again an outer loop over all strings and an inner loop over all characters is created. The HLS tool is immediately able to compile the kernel without the use of any pragmas. Because there are no bus requests, the minimum latency of the outer loop is much smaller; only 5 cycles. In this example, the iteration latency is improved by almost 10×. This means that our approach enables users to skip the tedious step of writing HLS-oriented C++ code to interface more efficiently with the data, while providing better performance at the same time. Developers are allowed to immediately focus on the computational aspect of the kernel.

```
#include <cstring>
#include <ap_int.h>
#include <hls_stream.h>
int filter_hls_normal(const int num_entries,
                      const int *in_first_name_offsets,
                      const char *in_first_name_values,
                      const int *in_last_name_offsets,
                      const char *in_last_name_values,
                      const int *in_zipcode,
                      const char filter_name[64],
                      const int filter_zipcode,
                      int *out_first_name_offsets,
                      char *out_first_name_values) {
#pragma HLS INTERFACE ap_bus latency=25 port=in_first_name_offsets
#pragma HLS INTERFACE ap_bus latency=25 port=in_first_name_values
#pragma HLS INTERFACE ap_bus latency=25 port=in_last_name_offsets
#pragma HLS INTERFACE ap_bus latency=25 port=in_last_name_values
#pragma HLS INTERFACE ap_bus latency=25 port=out_first_name_offsets
#pragma HLS INTERFACE ap_bus latency=25 port=out_first_name_values
  int matches = 0;
  // Name length buffers:
  int fn_strlen = 0; int ln_strlen = 0; int fn_offset = 0; int ln_offset = 0;
  char fn_buffer[64]; char ln_buffer[64];  // Name buffers
  int zip = 0;  // Zip code buffer
  int offset_index = 0; int offset_value = 0;  // Output buffers
  out_first_name_offsets[offset_index] = offset_value;  // Write first offset
  // Iterate over every entry:
  for_each_entry: for (int e = 0; e < num_entries; e++) {
    bool match = true;  // Assume a match
    // Get string lengths & zip
    fn_strlen = in_first_name_offsets[e + 1] - in_first_name_offsets[e];
    ln_strlen = in_last_name_offsets[e + 1] - in_last_name_offsets[e];
    zip = in_zipcode[e];
    // Copy over to buffers
    memcpy(ln_buffer, in_last_name_values + ln_offset, ln_strlen);
    memcpy(fn_buffer, in_first_name_values + fn_offset, fn_strlen);
    // Check if the last name matches
    match_last_name: for (int c = 0; c < ln_strlen; c++) {
      if (ln_buffer[c] != filter_name[c]) { match = false; }
    }
    // If it matches, write back into the name buffer.
    if_match: if (match) {
      matches++;
      memcpy(fn_buffer, out_first_name_values + offset_value, fn_strlen);
      offset_value += fn_strlen;
      offset_index++;
      out_first_name_offsets[offset_index] = offset_value;
    }
  }
  return matches;
}
```

Figure 4.3: Filter example in HLS not using streaming interfaces.

```
#include <cstring>
#include <ap_int.h>
#include <hls_stream.h>
int filter_hls_fletcher(int num_entries,
                        hls::stream<int> &in_first_name_length,
                        hls::stream<char> &in_first_name_values,
                        hls::stream<int> &in_last_name_length,
                        hls::stream<char> &in_last_name_values,
                        hls::stream<int> &in_zipcode,
                        char filter_name[64],
                        int filter_zipcode,
                        hls::stream<int> &out_first_name_length,
                        hls::stream<char> &out_first_name_values) {
  int matches = 0;
  int fn_strlen = 0; int ln_strlen = 0; // Name length buffers
  char fn_buffer[64]; char fn_char; // Name buffers
  int zip; // Zip code buffer
  // Iterate over each entry
  for_each_entry: for (int e = 0; e < num_entries; e++) {
    bool match = true;  // Assume the filter matches
    // Grab the lengths and zip code
    in_first_name_length >> fn_strlen;
    in_last_name_length >> ln_strlen;
    in_zipcode >> zip;
    // Buffer the names
    int fc = 0;
    for_get_fn: for (fc = 0; fc < fn_strlen; fc++) { in_first_name_values >> fn_buffer[fc]; }
    // Make sure to terminate the string
    if (fc != 63) { fn_buffer[fc + 1] = '\0'; }
    for_get_ln: for (int c = 0; c < ln_strlen; c++) {
      char lnc = '\1';
      in_last_name_values >> lnc;
      // As the characters stream in, check last name equal to filter name:
      match_name: if (lnc != filter_name[c]) { match = false; }
    }
    // Check the second filter condition: zip code
    match_zip: if (zip != filter_zipcode) { match = false; }
    // Only output the first names if the match was true
    if (match) {
      matches++;
      // Output the string length
      out_first_name_length << fn_strlen;
      // Output the characters
      for_put_fn:
      for (int c = 0; c < fn_strlen; c++) {
        out_first_name_values << fn_buffer[c];
      }
    }
  }
  return matches;
}
```

Figure 4.4: Filter example in HLS using streaming interfaces supplied by Fletcher.

4.6. Accelerating the PairHMM Forward Algorithm

Next-generation DNA sequencing methods allow cost-effective sampling of DNA [4]. This data is used e.g. to understand and treat human diseases. The analysis of the huge amounts of data resulting from such samples is still a computational challenge today. Hidden Markov Models (HMM) are used during analysis to find pairwise alignments of DNA sequences. More specifically PairHMMs [5] can be used to calculate the probability that two sequences are related, which is called the overall alignment probability. In this section, we consider the alignment probability of what is called a short-read to a haplotype, both very small sequences of DNA, although the read is typically shorter than the haplotype.

Because of the computational complexity and the data volume, PairHMM calculations in genome analysis pipelines (such as Genome Analysis ToolKit or GATK [6]) take a long time to complete on conventional machines. However, the PairHMM Forward Algorithm, which is also used in the software implementation of the GATK HaplotypeCaller, is an algorithm exhibiting a long datapath. Such algorithms are often good candidates for FPGA implementation. An FPGA accelerator is often able to achieve a high throughput and high power-efficiency. In other research, it has been shown that FPGAs can be suitable candidates to implement the algorithm using Systolic Arrays (SAs). However, a drawback of some architectures is that the computational resources are sometimes under-utilized due to control issues or data padding.

In this section, we attempt to optimize SA utilization, allowing for near continuous processing on all the computational elements of the SA. Our future aim is to implement many small but efficient SAs instead of implementing one large but inefficient SA. Our contributions are as follows:

- We provide a model to calculate the utilization of an SA.
- We analyze architectural alternatives allowing continuous processing of the PairHMM Forward Algorithm.
- We implement one such architecture that is more than 2.5x faster than the state-of-the-art FPGA implementation and 10x faster than a state-of-the-art CPU.

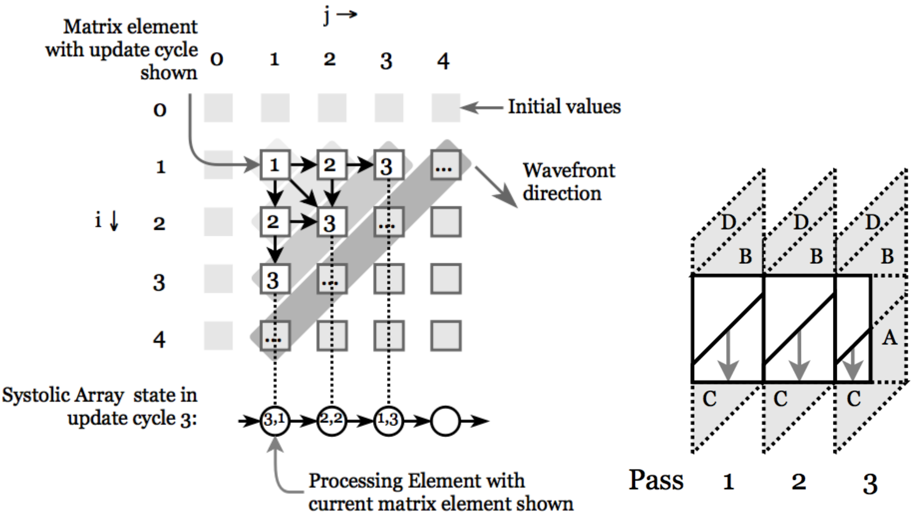
4.6.1. BACKGROUND

PAIRHMM FORWARD ALGORITHM

Algorithm 1 PairHMM Forward Algorithm used in the GATK HaplotypeCaller

```
$M \leftarrow I \leftarrow D \leftarrow 0_{X+1,Y+1}$
$D_{0,0...Y} \leftarrow C_{init}$
for $i \leftarrow 1, X$ do
    for $j \leftarrow 1, Y$ do
        $M_{i,j} \leftarrow \alpha_{i,j} \cdot (\beta_i \cdot M_{i-1,j-1} + \gamma_i \cdot I_{i-1,j-1} + \gamma_i \cdot D_{i-1,j-1})$
        $I_{i,j} \leftarrow \delta_i \cdot M_{i-1,j} + \epsilon_i \cdot I_{i-1,j}$
        $D_{i,j} \leftarrow \eta_i \cdot M_{i,j-1} + \zeta_i \cdot D_{i,j-1}$
    end for
end for
return $\sum_{j=0}^{Y} M_{X,j} + I_{X,j}$
```

The PairHMM Forward Algorithm as implemented in the HaplotypeCaller is seen in Algorithm 1. M, I and D are the matrices for match, insertion and deletion probabilities. $\alpha_{i,j}$ is the emission probability: for each position in the read i it can have two different values, depending on the bases of the read and haplotype at position i and j. β, γ, δ, ϵ,


Matrix element with update cycle shown
j →
0
1
2
3
4
Initial values
Wavefront direction
i ↓
1
2
3
...
Systolic Array state in update cycle 3:
3,1
2,2
1,3
Processing Element with current matrix element shown
D
B
A
C
Pass
1
2
3

(a) One pass detail

(b) Multiple passes and potential overhead

Figure 4.5: An example of how an SA can solve a PairHMM using the Forward Algorithm (Algorithm 1).

η and ζ are transmission probabilities that only depend on the read position i. In the software implementation, all probabilities are floating-point values. We define X and Y as the length of the read and haplotype, respectively.

When updating some cell (i, j) of the matrices M, I and D, a dependency exists on the values of cells $(i-1, j-1)$, $(i-1, j)$ and $(i, j-1)$. Thus, only matrix cells laying on the anti-diagonals of the matrix can be updated in parallel. Therefore, Algorithm 1 is commonly implemented in hardware using a one-dimensional systolic array (SA) consisting of a number of processing elements (PEs). Each PE implements the inner loop in the algorithm, updating one cell in each of the matrices M, I, and D. During every update cycle, the SA updates the cells on the anti-diagonal of the matrices (sometimes called a 'wavefront'). A simplified diagram of such an SA can be seen in Figure 4.5a. As the anti-diagonal grows, the amount of exploitable parallelism grows as well.

When the length of the haplotype (or read) is larger than the number of elements in the SA, the SA can compute the matrices by making multiple vertical (or horizontal) *passes* through the matrix, processing only a subset of columns (or rows) and wrapping back to the top (or side) of the matrix after completion of a pass. This can be seen in Figure 4.5b. The values in the last column (or row) in the pass are often stored in a FIFO buffer. Whenever a pass is shorter than the amount of PEs in the SA, padded data is inserted (Figure 4.5b case A).

RELATED WORK

Earlier research discussed using SAs to solve similar HMM-based algorithms in the field of computational biology [7][8]. These proposed SA designs introduce overhead when model parameters must be reconfigured between subsequent passes or workloads. Subsequent research such as [9] and [10] show more advanced SA designs, deploying double buffering of model parameters of alternating passes and workloads, allowing for near continuous processing.

More recent work implements the same PairHMM Forward Algorithm as this work in FPGA on the Convey Computer platform, showing higher throughput than single threads of the host processor[11]. However, the architecture introduces overhead when switching between passes, as parameters are shifted into the PEs. In [12], which we consider as the current state-of-the-art FPGA implementation, PEs are partially internally pipelined, achieving a high throughput. This design uses the CAPI interface of the IBM POWER8 platform, which we will also use in this work.

In this paper, we introduce a new architecture that is able to continuously perform *useful* calculations in the PEs of the SA. Once the first input data pair is loaded, our design wastes virtually no cycles due to memory latency or parameter reconfiguration. Thus, the design is able to achieve extremely close to the maximum theoretical performance of a fixed-size SA.

4.6.2. PERFORMANCE MODEL

We define the length of the read and the haplotype as X and Y. The total amount of cell updates required to process the Forward Algorithm is $X \times Y$. A useful measure of performance for the Forward Algorithm is the throughput in number of cell updates per second (CUP/s). In this paper, we will only count *effective* cell updates, which are cell updates that contribute to the final result (i.e. not on padded data).

The throughput of an SA design is affected by the average utilization of the PEs. We observe that while processing the Forward Algorithm with an SA, under-utilization of the PEs may be introduced in several cases (also shown in Figure 4.5b):

(A) When data is padded if a pass is not as wide as the SA.

(B) If the PEs in the SA may only work on one pass at a time, under-utilization of the PEs occurs at the start of a pass.

(C) Same as B, but at the bottom of a pass.

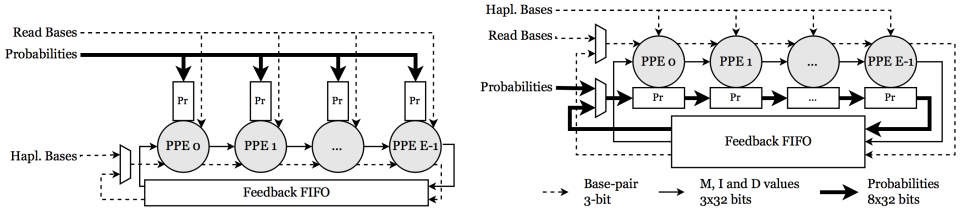
(D) When switching between passes, to update the model (α, β, etc.) in the PEs.

(E) When the height of the matrix is shorter than the number of PEs, and more than one pass is required, the read must be padded. Otherwise, the feedback FIFO will not contain any data yet for the first PE to work on in the next pass. (Not shown in Figure 4.5b).

We consider an SA of fixed size, thus the overhead introduced in case A and E is inevitable. However, we aim to eliminate the other causes of overhead.

FIXED-SIZE SYSTOLIC ARRAY PERFORMANCE

Consider the processing of the Forward Algorithm in an SA where; W is the width of the matrix, H is the height of the matrix and E is the number of PEs in the SA. Also, assume one cell update per clock cycle. In the ideal case, if we would process a large amount of pairs (thereby ignoring initial and final latency), that are of similar size, and if the input



(a) Architecture HS: haplotype data is streamed in horizontally, read data is streamed in vertically.

(b) Architecture RS: read data is streamed in horizontally, haplotype data is streamed in vertically.

Figure 4.6: Two SA architectures.

data is available at any time at the inputs of the PEs, the average utilization of the whole SA for one pair is given by:

$$\text{Avg. utilization} = \frac{WH}{E\lceil \frac{W}{E} \rceil \cdot \max(E,H)} \tag{4.1}$$

Eq. 4.1 takes the number of cells in the original matrices and divides this by the number of cells in the padded matrices. This gives the ratio of effective cell updates verses all cell updates (including padding). In the case of such a workload, we may obtain the average number of *effective* cell updates U_{avg} per clock cycle by multiplying the average utilization by the number of PEs in the SA:

$$U_{avg}(W,H,E) = \frac{WH}{\lceil \frac{W}{E} \rceil \cdot \max(E,H)} \tag{4.2}$$

Thus, cells padded to the bottom of the matrix (in each pass, only when $H < E$) and cells padded to the right of the matrix in the final pass are also taken into account.

If the height of the matrix is equal or larger than the number of PEs (i.e. $H \geq E$) and the width of the matrix is an integer multiple of the number of PEs (i.e., $W = nE, n \in \mathbb{Z}_{>0}$), all PEs perform useful work in every pass. In this case, maximum throughput is achieved ($U = E$). This also shows an SA of length $E = 1$ is always maximally efficient (i.e. an SA of this size needs no padding, since passes are of width 1).

Modern FPGAs contain enough computational fabric to implement a large number of PEs. However, the number of SAs cannot be as high, since it quickly becomes bounded by the available memory and interconnect. For example, the FPGA used for this work offers enough resources to implement 112 PEs, but the FPGA lacks resources to implement 112 SAs in parallel, requiring 112 controllers, input buffers, feedback FIFOs and other items in the data and control paths. A more feasible combination would be to have, e.g. 7 SAs of 16 PEs each. This work focuses on implementing an architecture for a single SA, that achieves as close to the maximum performance of Eq. 4.2 as possible.

4.6.3. ALTERNATIVE ARCHITECTURES

To achieve the maximum performance, the matrix can be mapped onto the SA in two ways. In one, (HS in Figure 4.6a), the data that depends on the haplotype position (haplotype bases) is streamed-in at the head of the SA. The data that depends on the read position (probabilities and read bases) is fed vertically into the PEs. In this approach, the matrix is mapped to have the read on the horizontal axis, and the haplotype on the vertical axis of the matrices. The other approach (RS, Figure 4.6b) has horizontal and vertical data streams swapped.

All data that is fed *horizontally* can be streamed from input FIFOs into the head of the SA. When reuse of this data is required in a new pass, the feedback FIFO will provide this data and intermediate values that were streamed out of the SA after processing the last column of the previous pass. All data that is fed *vertically* can be distributed to the respective PEs using a bus connected to registers (or RAM).

Although architectures similar to HS are often used (with the exception of [9]), we argue for the use of RS. The reason to select RS is related to the sizes of the read and haplotype, X and Y. The haplotype is at least as long as the read, but often much longer. Consider again Eq. 4.2. When the ratio between fully utilized passes and underutilized passes is high (i.e. when Y is large) the efficiency is also high, since a relatively larger number of passes will have full SA utilization.

Internally, the PEs are pipelined, such that the critical path in the circuit is reduced, allowing higher clock frequencies for the whole SA. The throughput of the SA is directly proportional to its clock frequency.

4.6.4. MAXIMIZING UTILIZATION

To achieve maximum utilization, overhead from the cases B, D and C described in Section 4.6.2 must be prevented. This can be done by observing that, during one cell update cycle, the vertical data of *at most* one PE needs to be updated, i.e. at most one PE in the SA will enter a new pass in each cell update cycle. Therefore, a bus connected to the vertical data registers needs to transfer the vertical data of only one PE per cycle.

In this way, any data that is still in the SA from a previous pass or pair does not have to be completely streamed out, allowing cell updates between passes and pairs to take place within the SA (solving case B and C). Furthermore, when the vertical data bus is able to transfer all required data in one cycle, overhead caused by updating model parameters in the PEs can be avoided (solving case D)

An example of continuous processing on the RS architecture is given for the following case: The number of PEs, $E = 4$, the length of the read $X = 6$, the length of the haplotype: $Y = 6$, the read is 'GTACAT' and the haplotype is 'ACTGTC'.

As shown in Figure 4.7, on each anti-diagonal, the state of the complete SA is depicted during one cell update cycle, and superimposed over the matrix cells of a pass. For each cell update cycle, the vertical data of *at most* one PE must be updated. Similarly, the output of at most one PE holds data contributing to the final result. Therefore, the M and I output of each PE are logically OR-ed with each other and sent to an accumulator. This implements the last line of the procedure in Algorithm 1. By setting the haplotype and read base to a value called "Padding" (denoted by 'P' in the figure), the PEs output will be invalidated.

CONTROL MECHANISM

Since PEs are internally pipelined (Section 4.6.3), to allow multiple PairHMMs to run in each of the pipeline slots, one could use BRAM and allocate a specific region for each

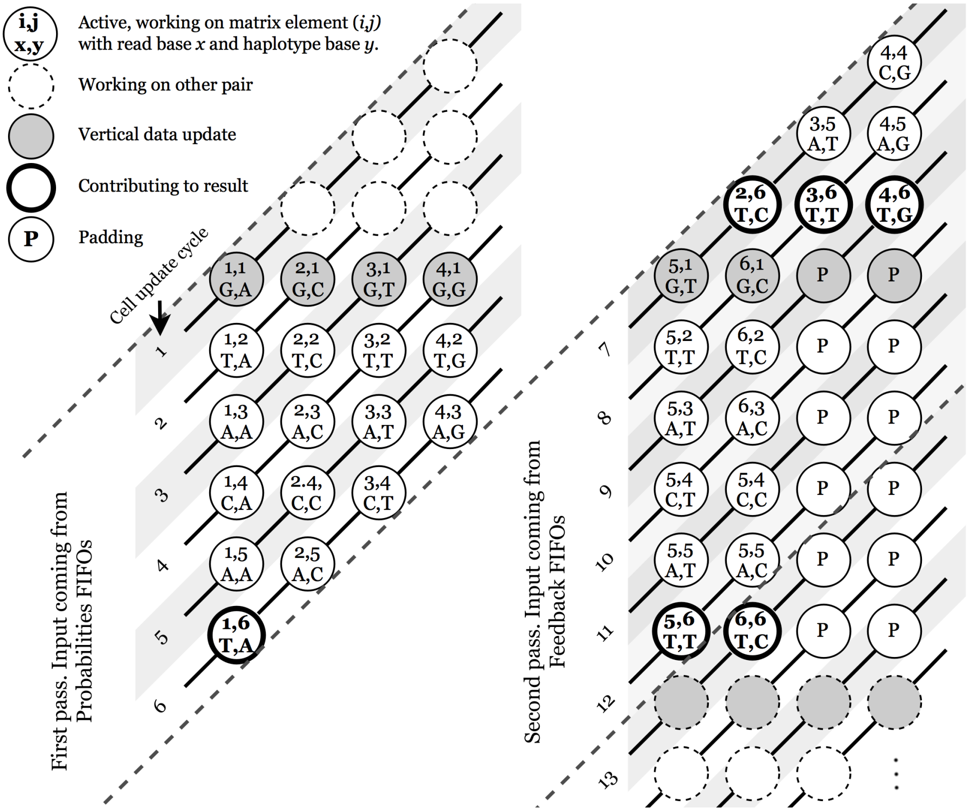


Figure 4.7: Example of processing a pair for which the read length $X = 6$, the haplotype length $Y = 6$ and the number of PEs $E = 4$.

of the N pairs that is active in an N stage pipeline. However, such a control mechanism is complex, since it must track all SA control signals, as well as RAM addresses, for each of the N pipeline slots independently. At the side of the memory interface, it must keep track of N pointers, data counters, and other control information.

The control mechanism can be extremely simplified by allowing the smallest unit of processing to be *batches* of N pairs. By implementing FIFOs for the input data, the host can prepare a batch of N pairs to be processed, ordering the batch in memory in such a way that the accelerator itself does not have to deal with ordering at all. The accelerator keeps track of control signals of only one batch instead of keeping track of all control signals for each of the N pairs.

Although simplifying control complexity, batches have a minor drawback in terms of performance; if the pairs contained in the batch are of completely different sizes, smaller pairs require a lot of padding, in turn decreasing SA efficiency.

Consider the processing of N pairs in a batch, where the n-th pair has read length X_n and haplotype length Y_n. The total amount of work required in cell updates U_{req} to process the batch is given by:

$$U_{req} = \sum_{n=0}^{N-1} X_n Y_n \tag{4.3}$$

When the amount of work done on a batch U_{batch} is determined by the largest read and haplotype, it can be calculated (containing overhead due to padding) using Eq. 4.1 as follows:

$$U_{batch} = N \cdot \max(\max_n X_n, E) \cdot E \left\lceil \frac{\max_n Y_n}{E} \right\rceil \tag{4.4}$$

Dividing Eq. 4.3 by Eq. 4.4 gives the efficiency per batch.

When the read and haplotype lengths are different, the SA has low efficiency due to abundant padding. A large portion of this drawback can be mitigated by sorting the pairs by number of passes required, then sorting each list of pairs with the same number of passes by read size. After sorting, the batches are created by the host and sent to the accelerator. When the workload is very large, sorting makes it likely that haplotypes and reads inside a batch share a similar number of passes and read size.

To reduce the sorting time, we sort only small subsets of the workload. For the whole genome sequencing dataset we used for this work (see Section 4.6.6), we split the workload into 1832 subsets of 2^{14} pairs and sort them. In Figure 4.8, we compare it to the SA utilization when using unsorted subsets and the ideal utilization given by Eq. 4.2, in the case where we would not use batches, but are able to start working on pairs in independent pipeline slots. We find that using sorted batches almost achieves ideal performance.

4.6.5. IMPLEMENTATION

We implemented architecture RS using an AlphaData ADM-PCIE-7V3 FPGA accelerator card, for which a POWER8 CPU on an IBM Power System S824L (8247-42L) serves as a host. This system offers the Coherent Accelerator Processor Interface (CAPI) to the accelerator through IBMs Power Service Layer (PSL) interface. The memory interface at the host side is therefore similar to [12]. To abstract away the PSL interface, we use the CAPI Streaming Framework from [13].

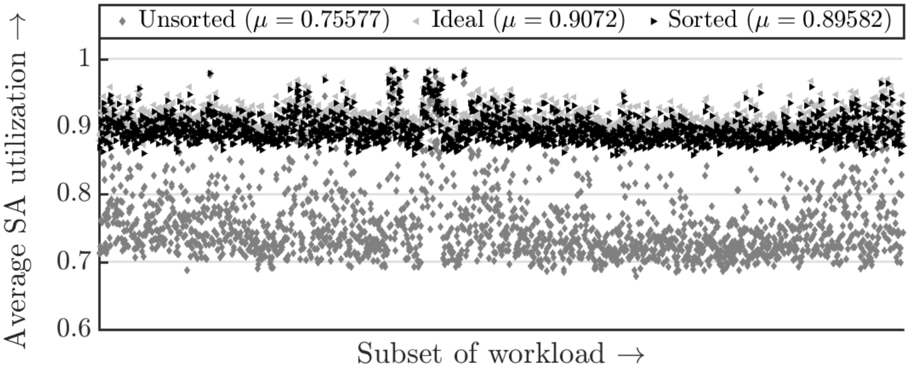


Figure 4.8: Effect of sorting on the efficiency of the SA, with E=16.

The SA consists of E Pipelined Processing Elements (PPEs). Each PPE implements the inner loop of Algorithm 1 as a 16-stage pipeline. The maximum number of PPEs we could fit (using Vivado 2016.2) was 112. This bound is determined by the number of DSP blocks. The DSP blocks are used by the floating-point units in the PPEs. The FPGA allows 3600 DSP blocks to be used, but the PSL is distributed as a pre-routed design and prevents the use of a quarter of the DSP blocks. In this work, we implement the SA using $E = 16$ and $E = 32$.

4.6.6. Experimental results

To measure the performance for different sizes, we generate workloads of increasing read (X) and haplotype (Y) size, where $Y \geq X$, in steps of 4. Each workload contains 2^{14} pairs. The performance for each workload is shown in Figure 4.9. Our SA runs at 166.7 MHz, thus the maximum theoretical throughput is $E \cdot f$ in cell updates per second (CUP/s).

Padding in the horizontal direction (when $X < E$), deteriorates the throughput, as the utilization of the SA is very low. When there is no padding in the horizontal direction, the throughput quickly grows towards the maximum theoretical throughput. Also, the effect of having haplotype sizes of integer multiples of the number of PEs is clearly visible. In this case, the performance nears the maximum theoretical throughput. The highest throughput measured was 99.76% of the maximum. The last bit of overhead is introduced by the memory latency at initialization and termination.

For a realistic benchmark, we use the same dataset as the work presented in [12] (whole human genome dataset G 15512.HCCI954.1 mapped to chromosome 10). The dataset contains over 30 million pairs. We split and sort the dataset in subsets of 2^{14} pairs. The results for sizes $E = 16$ and $E = 32$, the maximum theoretical throughput for each SA, the reported throughput of [12] and [11] and the reported maximum for the POWER8 host CPU are shown in Figure 4.10.

For $E = 32$, we achieve a throughput of 84% of the maximum performance; for $E = 16$, this is 93%. The lower throughput for $E = 32$ is caused by the large number of reads in the dataset of which the size is smaller than E, resulting in much variation. However, for the SA with $E = 16$, we observe that the utilization is higher, since padding occurs

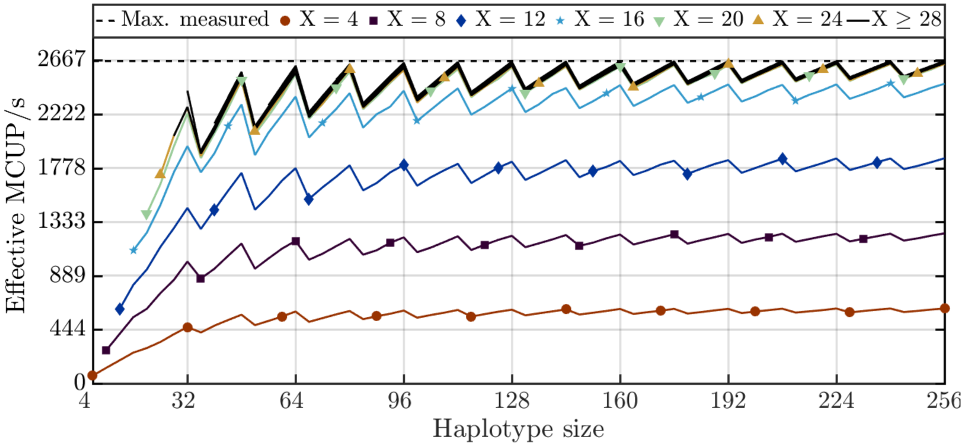


Figure 4.9: Synthetic benchmark. PEs: $E = 16$. Workload size: 2^{14}. Step size: 4. Read size: X. Theoretical maximum throughput: 2667 MCUP/s. Max. measured: 2661 MCUP/s.

Table 4.5: FPGA post-routing power estimate and area

Part	**LUTs**	**Registers**	**RAM36**	**DSP**	**Power(W)**
Available: 7VX690	433200	866400	1470	3600	
16 PEs + interfaces	119937	140397	473	378	11.212
16 PEs, this work only	47346	60525	181	354	2.721
32 PEs + interfaces	163450	189085	473	730	13.213
32 PEs, this work only	90862	109213	181	706	4.585

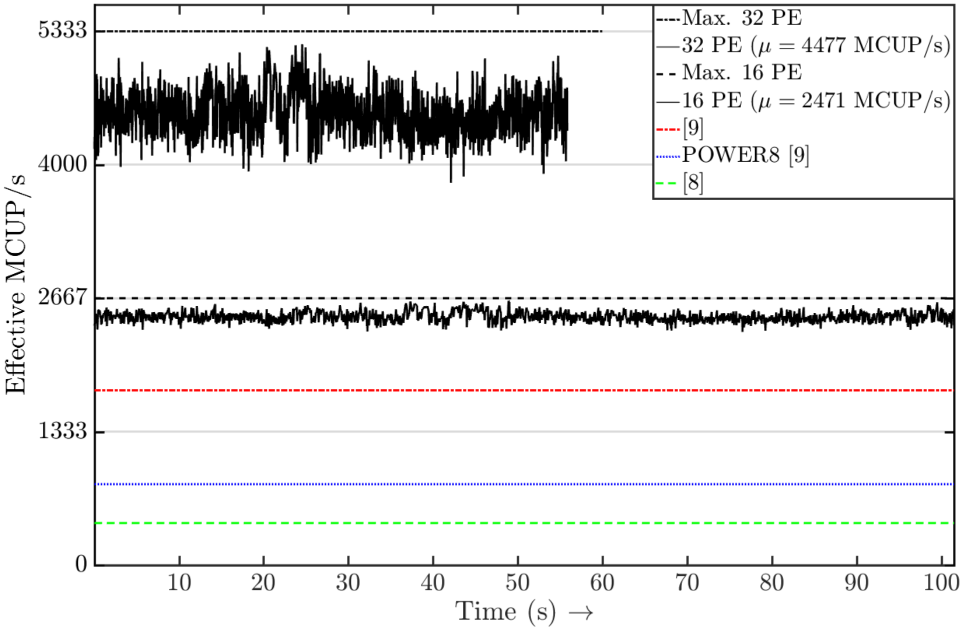


Figure 4.10: SA throughput using a real dataset with $E = 16$ and $E = 32$. Subsets size 2^{14}

less. Although for $E = 32$, the SA is twice as long as for $E = 16$, the run-time is only 1.8x lower. Furthermore, with the same amount of processing elements, our architecture shows an average improvement of throughput of 2.5x over the state-of-the-art. With half the processing elements, our implementation achieves a 1.4x higher throughput.

In Table 4.5 the area statistics of the SA design with 16 and 32 PEs are shown after placing and routing. We show the logic available in the device, the logic utilization of our system (including interfaces) and for our design only. Moreover, the power estimation of Xilinx Vivado is included. From Table 4.5 and Figure 4.10, we estimate the power efficiency to be $339 \cdot 10^6$ CUP/J.

4.6.7. Pair-HMM Posit Accelerator

The architecture of the streaming-based pair-HMM accelerator described in [14] and the previous section, is based on a widely-used software implementation in [15]. We improve the design to make use of Arrow tabular data sets, leveraging interfaces generated through Fletcher, and allowing the design to be integrated efficiently in all the supported software languages.

We furthermore exploit the capabilities of the FPGA accelerator to implement the arithmetic of the design using posit units rather than IEEE floating-point arithmetic. Posit arithmetic can only be emulated on general purpose processors, since at the time of writing, no commercially available posit-enabled processors are available. Posit arithmetic is floating-point arithmetic with a different type of lay-out, that in some cases may improve numerical precision of results. By using posit arithmetic and by avoiding intermediate

	Haplotypes
	haplo (8-bit)
0	base pair
	…
	base pair
1	base pair
	…
	base pair
⋮	…

	Reads	
	read (8-bit)	probabilities (256-bit)
0	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
	…	…
	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
1	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
	…	…
	base pair	$\alpha_{\text{diff}}\ \alpha_{\text{simi}}\ \beta\ \gamma\ \delta\ \epsilon\ \eta\ \zeta$
⋮	…	…

Table 4.6: Schematic overview of the Arrow schema for the Arrow pair-HMM Accelerator implementation, consisting of the columns used to feed the pair-HMM accelerator.

rounding, the precision of the final results is improved. The details of the posit arithmetic units that have been developed are described in more detail in [16].

As described in the previous section, the input to the accelerator consists of a set of haplotype base pairs, read base pairs and the emission and transmission probabilities related to these reads.

The Arrow data set designed for this implementation is depicted in Table 4.6. As can be seen, the data set consists of two separate tables used to represent the haplotypes as well as the reads for a specific batch. The haplotype and read base pairs are represented by an 8-bit wide field, being able to represent any ASCII character. For each read, the emission and transmission probabilities for this read are located in the second column of this table and are represented as a single columns but they are grouped through Arrow's struct type. The probability α can contain a penalty if the read and haplotype base pairs are not equal during the pair-HMM forward algorithm evaluation. Hence, two values for this probability are stored. As there are always eight emission and transmission probabilities in total, the width of this column is equal to 256 bits, as each probability is represented by a 32-bit posit number. The type of the column entries can therefore be chosen as a fixed-size primitive.

The entry index indicated in the diagram represents the batch to be processed by the accelerator. The accelerator is able to access specific batches based on this index, as will be illustrated later. As the amount of base pairs inside one batch is variable, the length of each entry is also variable. When an entry is read by the accelerator, it also receives the length of this entry.

A schematic overview of the high-level components of this pair-HMM accelerator design is depicted in Figure 4.11. For the input basepair reads, a Array Reader is used in order to read the base pairs and emission/transmission probabilities from the data set. A second Array Reader is instantiated to read the basepairs from the haplotype data set. The incoming streams are controlled by a scheduler that makes sure the input data is fed into the systolic array in the correct cycle. The posit fields of the input probabilities, represented as 32-bit posit numbers, are extracted using the posit extraction unit as described in [16].

The outgoing calculation results from the systolic array, being raw posit values with unrounded fraction fields, are then normalized. The normalized 32-bit posit words are fed

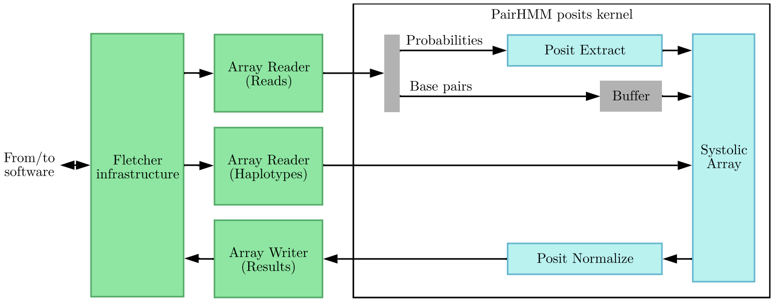


Figure 4.11: Schematic overview of the high-level components inside the pair-HMM accelerator core design, interfacing with Apache Arrow.

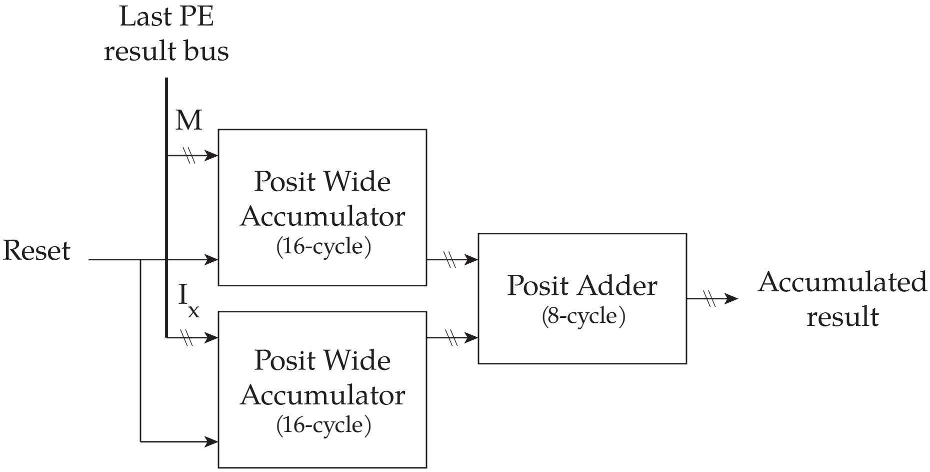


Figure 4.12: Schematic overview of the pair-HMM accumulation stage using posit wide accumulator units.

into a Column Writer in order to write the results into an Arrow data set in host memory.

ACCELERATOR MICROARCHITECTURE

The architecture proposed for this implementation is based on a fixed-size systolic array design that is optimized for maximum pipeline utilization as described in the previous section, although all arithmetic units are replaced by their posit counterpart. Quickly specializing the PairHMM circuit to use an arbitrary type of numerical representation leverages the reprogrammable capability of the FPGA accelerator, something unique to this accelerator platform.

In contrast to the original design of the previous section, for the posit design, the intermediate results of calculations performed inside a Processing Element (PE) are kept unrounded whenever possible. The purpose is to improve the overall decimal accuracy of the final likelihood computation results produced by the pair-HMM accelerator by means of the forward algorithm. The elements of the last row in the *M* and *I* matrix are added and accumulated for each column. These matrix elements are calculated by the last PE in

Config		Available	Used (core)		Used (total)	
posit(32,2)	**LUT**	331680	185174	(55.83%)	264078	(79.62%)
	Register	663360	179229	(27.02%)	271031	(40.86%)
	BRAM	1080	99	(9.17%)	425	(39.35%)
	DSP	2760	704	(25.51%)	723	(26.20%)
	Power		18.299 W		25.379 W	

Table 4.7: FPGA resource utilization and power consumption estimation of the pair-HMM posit accelerator implementation for Apache Arrow, both for the accelerator core only and for the total implementation including the Power Service Layer.

the systolic array design. In order to maintain as much accuracy as possible, our design uses wide accumulators, also shown in Figure 4.12. For each matrix of the pair-HMM forward algorithm a separate wide accumulator sums every column of its last row. The latency of a posit accumulator unit in terms of number of cycles is equal to the depth of the systolic array (16 PEs) because each matrix element is calculated per pair, thus allowing up to 16 pairs to be computed per pass through the systolic array. Therefore, the accumulated value for a given pair is updated every 16 cycles when new matrix elements for this pair are computed.

The advantage of using wide accumulators is that more information is kept while accumulating the matrix elements of the forward algorithm. Implementing this design in the pair-HMM accelerator will result in a longer critical path in the internal circuit. Since more logic is needed in order to process the wider fractions of accumulated values, this affects either the clock frequency or latency of the design. Therefore, the decision whether to integrate the wide accumulator design into an overall accelerator design depends on a trade-off between performance and precision.

EVALUATION

An implementation of a single pair-HMM accelerator core has been generated and tested for the *posit(32,2)* configuration. We analyze FPGA resource used, decimal accuracy [17] of calculation results, and throughput performance as well as speedup compared to software implementations of the pair-HMM algorithm. The machine used in these experiments is the IBM Power Systems S822LC featuring two 10-core POWER8 CPUs running at 2.92 GHz. This machine is equipped with the Alpha Data ADM-PCIE-KU3 accelerator card featuring the Xilinx Kintex UltraScale XCKU060 FPGA used for this design.

Table 4.7 shows the area utilization statistics for the posit dot product accelerator implementations, along with estimated power consumptions. The power consumption for only the accelerator core as well as for the total design is displayed. The overall design includes the Fletcher-generated logic and the Power Service Layer (PSL), required for interfacing with the host using CAPI.

Figure 4.13 shows the decimal accuracy of the calculation results produced based on simulation of the proposed hardware pair-HMM accelerator. The decimal accuracy of the *posit(32,2)* hardware implementations is evaluated, together with a software evaluation of the pair-HMM forward algorithm using the *float* format.

The reference calculation for determining the decimal accuracy is performed in a 100-decimal accuracy number format using the Boost Multi- precision C++ library, providing

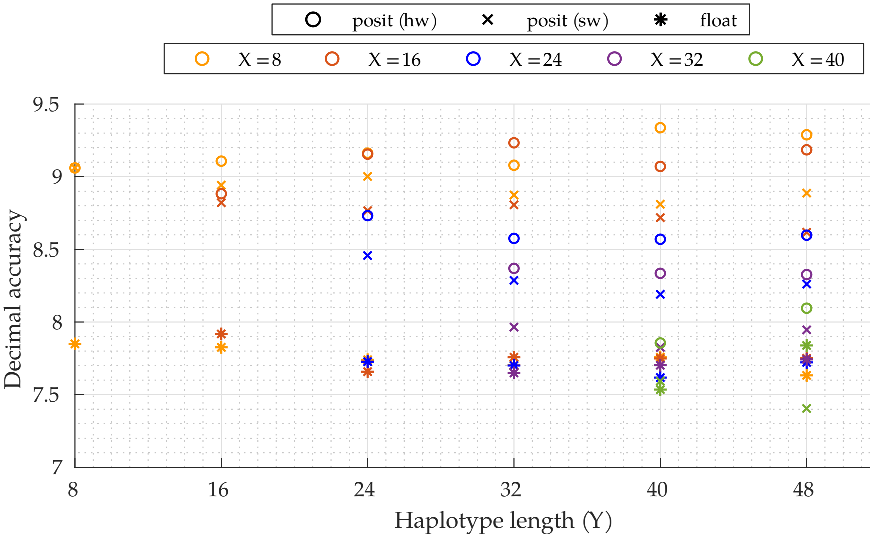


Figure 4.13: Decimal accuracy of the proposed pair-HMM hardware accelerator results, compared to traditional *float* computation for *posit(32,2)* . X and Y denote the read and haplotype input sequence lengths respectively.

a number type with a customizable number of decimal digits of precision at compile-time [18].

For the presented evaluations, different combinations of input sequence lengths X and Y have been tested. The initial scaling constant is set at 2^{10}. For these conditions, both the software and accelerator calculation results are performing better than the traditional *float* format for nearly every test case, with an increase in decimal accuracy ranging between approximately 0.5 and 2 decimals of accuracy.

Appropriate caution should be taken with regard to the presented results. All pair-HMM forward algorithm calculations heavily depend on the initial conditions. These conditions are, apart from the input read/haplotype bases and emission/transmission probabilities, influenced by the chosen initial scaling constant. The comparison of different initial scaling constants and their effect on the decimal accuracy of final calculation results as depicted in Figure 4.14 shows this behavior, along with the proof that scaling constants exist that result in better decimal accuracy compared to the best achievable decimal accuracy for the *float* format.

The average performance for the pair-HMM hardware accelerator interfacing with the Apache Arrow columnar memory format implementation in terms of MCUPS for different combinations of sequence lengths X and Y is depicted in Figure 4.15a. This performance benchmark is performed for 2^{15} base pair comparisons. As can be seen, the throughput decreases for any input sequence length X lower than the number of PEs in the systolic array due to under-utilization of the overall accelerator. The theoretical maximum throughput of 2000 MCUPS is not fully reached due to the present hardware overhead. The explanation for this is that batch data is loaded into the accelerator buffers

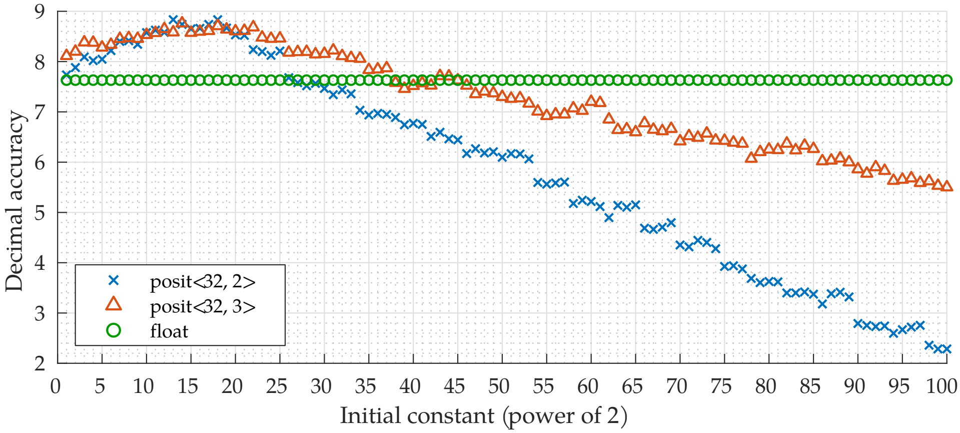
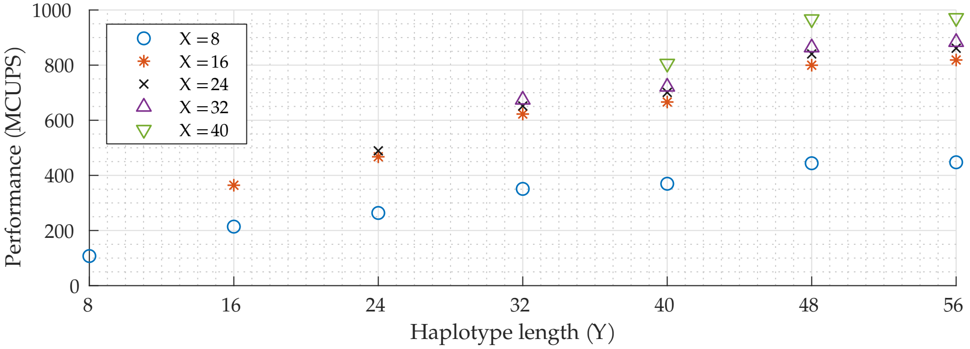


Figure 4.14: Decimal accuracy as a function of the initial scaling constant.

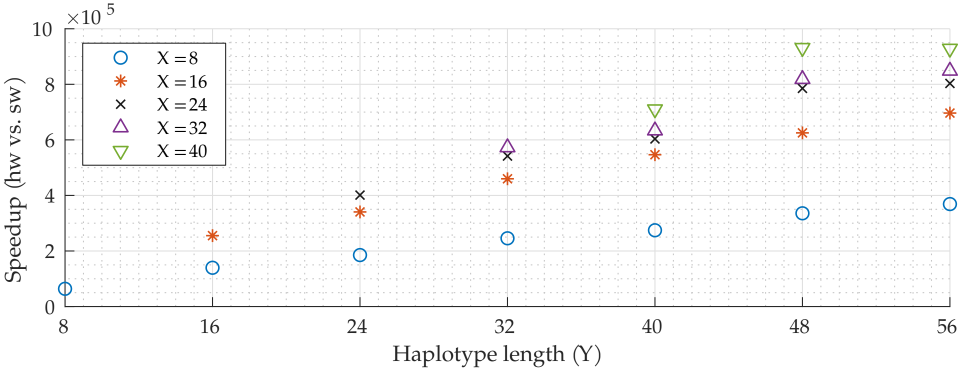
between batches, and the next batch will be loaded after finishing the previous batch. The overhead between initiating the read request to the host and receiving the full data set decreases the maximum achievable performance. The speedup of the pair-HMM hardware accelerator calculations compared to calculation in software (using a posit format emulation library) is depicted in Figure 4.15b for the same data sets. A significant speedup is observed for all tested combinations of read and haplotype input sequence lengths, ranging from a factor of approximately 10^5 to 10^6 times speedup.

4.6.8. SUMMARY

We analyzed the efficiency of systolic arrays that implement the PairHMM Forward Algorithm to find the overall alignment probability of a read to a haplotype. We have showed architectures that can implement fixed-size SAs in such a way that the overhead is minimal. We implemented one of the architectures, where the data corresponding to the read position is streamed through the systolic array. This implementation achieves 99.76% of the theoretical maximum performance for a synthetic dataset, and around 90% for a real dataset, depending on the size of the systolic array and the read-haplotype pairs. A systolic array with 32 processing elements is able to calculate the overall alignment probabilities of a whole genome dataset mapped to chromosome 10 in under 60 seconds, while only using approximately one third of the FPGAs DSP resources. We have upgraded the design to make use of Fletcher's Array Readers and Writers and have described the Arrow Schema that may be used to represent the structured data. This allows for efficient integration into software that is supported by Arrow. We have also leveraged the reconfigurable nature of the FPGA accelerator to drop-in replace the traditional floating point arithmetic units with posit arithmetic units. We stipulate that the design and implementation of this architecture is possible in a relatively short amount of time and low cost (compared to e.g. ASIC), because of the great flexibility that FPGA accelerators provide to customize the digital circuit according to the needs of the application. In future work, we aim to implement several small SAs in parallel, such that each SA may achieve a high utilization,

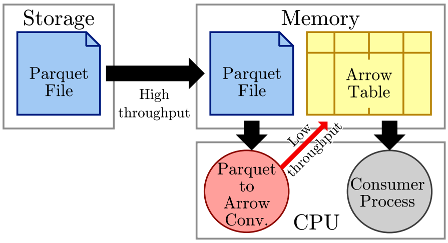


(a) Performance in terms of throughput (in MCUPS).

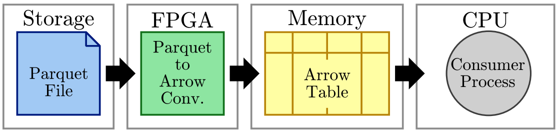


(b) Speedup of hardware versus software, based on total execution time.

Figure 4.15: Performance in terms of throughput (in MCUPS) and speedup compared to software calculation for the proposed pair-HMM accelerator design. X and Y denote the read and haplotype input sequence lengths respectively.



(a) Because I/O bandwidth has drastically increased, ingesting Parquet files causes the CPU to be the new bottleneck in big data processing pipelines.



(b) To alleviate the bottleneck, a heterogeneous system with an FPGA accelerator is proposed, where the FPGA accelerator performs an ingress transform of the stored file.

Figure 4.16: FPGA acceleration of the Parquet-To-Arrow converter

increasing the overall throughput.

4.7. Converting Apache Parquet to Arrow

In the context of big data analytics, the bandwidth associated with reading data from persistent storage is increasing rapidly due to the availability of non-volatile memory solid-state drives (e.g. NVMe SSDs). In the past, database systems were often designed with the assumption that *CPUs are fast* and *I/O is slow*. However, this relationship is quickly turning around over recent years. CPUs are no longer able to parse, decompress, and deserialize files at data rates close to I/O bandwidth, sometimes lacking over an order of magnitude in performance. While new storage formats designed with contemporary technology in mind may partially alleviate such bottlenecks, some fundamental limitations of performing decompression and deserialization with general purpose CPU's remain.

In order to improve the performance of database systems, we propose performing part of the decompression and deserialization of files to in-memory data structures with an FPGA accelerator (as shown in a contextual overview of this work in Figure 4.16). FPGA accelerators provide the following benefits within this context.

First, due to the excellent I/O capability of FPGA devices, it is possible to place the FPGA on the data path from storage to memory. Commercial FPGA accelerator cards with interfaces to SSDs are readily available today from various vendors. Second, because they are not limited by the drawback of a CPU's load-store architecture, FPGA systems can implement specialized data-flow designs with long pipelined data paths to perform the multitude of data movements required during the conversion. These are relatively inexpensive to implement in FPGA fabrics.

In this section, we contribute a design of an FPGA accelerator that takes files with large tabular data structures encoded in the well-known and widely-used Apache Parquet file format as input. The accelerator then converts these files into tables according to the Apache Arrow format in memory. This allows the FPGA accelerator to be leveraged in over 11 software languages. In Section 4.7.1, we describe related work and the Parquet and Arrow formats. We present the design and implementation of the accelerator in Section 4.7.2. Several performance and resource utilization characteristics of the accelerator are described in Section 4.7.3. We conclude this contribution in Section 4.7.4.

4.7.1. Background

In this section, we briefly discuss related work: the Parquet storage format used to convert from, and the Arrow in-memory format used to convert to.

Related Work

Previous research has acknowledged CPU processes to become the new bottleneck in big data processing pipelines, because I/O bandwidth is increasing [19][20][21][22]. An analysis of this problem specific to Parquet and ORC, and a proposal of an improved format is presented in [23], although the format results in a lower compression rate, the implementation is not freely available or widely used at the time of writing. In more recent work [24], the bottleneck is acknowledged, and FPGA-based solutions are provided at the level of the file system itself. The limitation also holds for network I/O, relevant to this paper in case distributed file chunks are shuffled, which is discussed in [25]. Previous work on reducing data duplication explored a specific combination of FPGA accelerators and Apache Parquet files [26]. Because the target throughput is still in the order of what contemporary CPUs can deliver, only the duplication reduction algorithm, but not the Parquet decompression and decoding itself are accelerated in an FPGA. At higher I/O bandwidths, decompression and decoding will form a bottleneck, which we will demonstrate and alleviate through the accelerator implementation of this work.

Apache Parquet

Parquet [27] is a storage format intended to store large tabular data structures in a column-oriented format, often used in distributed environments. To provide support for its multitude of features, each Parquet file has a complex hierarchical structure described by metadata in the footer of the Parquet file. This metadata describes the data types of the columns, and what compression and encoding schemes are used. The data itself is divided over *row groups*, containing one chunk of each column in the table, useful for distributed storage systems. The size of these row groups can be set when writing the Parquet file to allow for longer sequential reads in the same column chunk. The columnar format can be advantageous, e.g. only the relevant columns required by some computational transformation need be accessed without having to decode irrelevant columns. Column chunks are in turn divided into *pages*. Each page is compressed according to a specific compression codec, and its values are encoded using a specific encoding scheme. The locations of the pages and column chunks are found in the file footer. Every page can be independently decompressed and decoded, such that every page can be randomly accessed and processed in parallel.

APACHE ARROW

Typical for the big data framework ecosystem, Parquet is used in the context of a wide variety of software languages and run-times. When implementing a fast converter from Parquet files to in-memory data structures, it must be decided what (software) language and run-time engine will be at the consuming end of the data. The choice for a specific language, e.g. C++, rules out immediate use in another language, e.g. Python, unless one would perform the tedious work of implementing wrappers and/or serializers/deserializers. Fortunately, the Apache Arrow project provides a *common data layer*, where the in-memory representation of large tabular data structures is the same for any of the 11 supported languages. The project furthermore provides language-specific API's to access the data [28]. Applications in any of the supported languages may immediately benefit from an accelerated implementation of the conversion when the output is in the Arrow format.

Furthermore, we have created an FPGA accelerator framework built on top of Arrow, called Fletcher [29], which is used in this work. Fletcher generates DMA engines with streaming dataflow interfaces to and from Arrow in-memory data structures. The interfaces are generated based on Arrow *schemas* — descriptions of the data types of the values in the columns of the tabular data structures. The framework is more thoroughly discussed in Chapter 3.

4.7.2. DESIGN AND IMPLEMENTATION

CHALLENGES

The complex dynamic structure of a Parquet file makes it challenging to implement in hardware. Because the typical size of row groups is in the order of hundreds of MBs to multiple GBs, the overhead of parsing row group or even column metadata on CPU and performing host-to-accelerator communication is relatively small, and does not pose any bottleneck so far. Pages, however, are in the order of megabytes (the default is one MiB, although they can be chosen to be much larger). Therefore, we implement page metadata parsing in hardware as well as decompressing and decoding values in pages.

Even at this level of the data structure, the Parquet format offers challenges. A Parquet page itself consists of four distinct, variable-length, blocks of data. First, a header with page metadata, serialized according to Apache Thrift's Compact Protocol. Second and third, blocks containing the so called repetition and definition levels. These blocks are the result of Parquet optionally using Dremel encoding [30] (after Google's first implementation of this technique). These are used for nested data types (e.g. lists of lists) and/or nullable types. The last block contains the actual values. Because values of columns are stored contiguously and have the same data type, encoding techniques such as (among others) delta encoding with binary packing are used. They can also be compressed with compression codecs such as (among others) Snappy [31] and gzip.

We also find the following tenets important during the design. First, the converter should perform its function in a streaming fashion. This allows for an ingress-style transformation as shown in Figure 4.16b and keeps latency low, since there are no copies required on the on-board DDR memory of the accelerator card. Second, the converter should perform its function at a throughput close to the I/O bandwidth on either side of the accelerator. That is, either the SSD interface or accelerator to host memory interface.

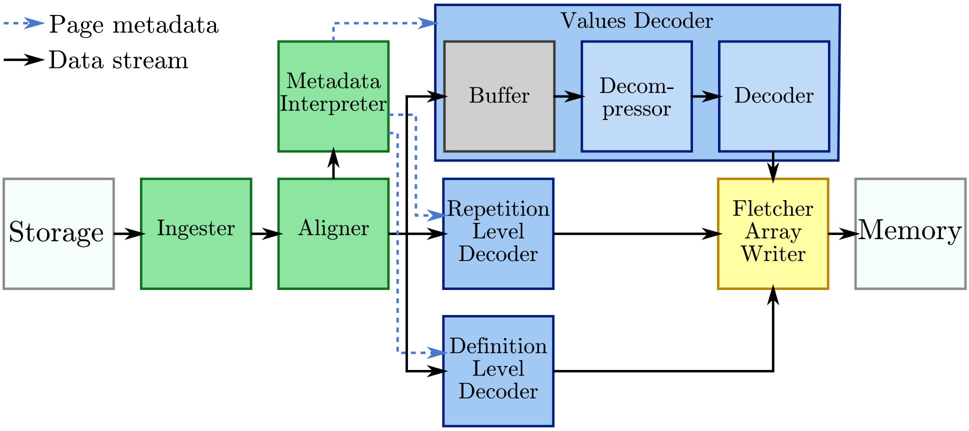


Figure 4.17: Architectural overview of the proposed accelerator. Control flow is omitted for clarity.

For contemporary and near-future systems, this is in the order of tens of GB/s.

Architecture

To solve the aforementioned challenges, we propose the top-level architecture of the Parquet-to-Arrow converter as shown in Figure 4.17.

It consists of the following components. The first three, Ingester, Aligner and Metadata Intepreter are always the same and required to convert any Parquet file. The implementation of the Values Decoder, Repetition Level Decoder and the Definition Level Decoder, depend on the compression and encoding scheme used by the file.

Ingester: At the request of the consuming software process, the Ingester initiates the loading of pages from memory or storage in large bursts. It produces two streams with raw bytes, and initial Parquet page alignment information within the raw byte stream, since every page is not necessarily aligned to the start of the stream that transports multiple bytes per transfer. These streams are fed into the Aligner component.

Aligner: Taking the streams from the Ingester, the Aligner implements a pipelined barrel shifter to position the raw bytes for the next stages. Because one of the three variable-length blocks within a page may be aligned differently, but could start within a streamed word of the previous block, the Aligner holds the unaligned words in a history buffer to be able to immediately restart the pipeline for the next page, without having to request the data again from memory or storage. The downstream components report back the amount of used bytes to provide the necessary control information for this functionality.

Metadata Interpreter: Once the page has been aligned, its metadata must first be interpreted. This is done by the Metadata Interpreter component. Parsing the metadata involves a rather complex state machine, because it must implement the used features of the Apache Thrift serialization protocol. This protocol uses rather dynamic features, such as variable-length integers, causing the metadata interpreter to absorb one byte per cycle. Because the page metadata is only a fraction of the total page data, the overhead of this relatively low-throughput process is negligible. After interpreting the header, the compressed and uncompressed size of the page and the number of values are known and streamed to the appropriate parts of the design.

Fletcher Array Writer: The Fletcher Array Writer is a component generated by the Fletcher framework, serving as a DMA engine that can write from hardware streams to in-memory arrays of complex data structures (e.g. nested lists) formatted by the Apache Arrow format specification. It must be noted that in this context, Arrow arrays are not C-like arrays, but can consist of multiple buffers holding data with specific relations expressed through the Arrow type. We feed the various streams emerging from the decoding of the values, and the repetition and definition levels, into the ArrayWriter. In turn, it will write the data into memory in the Arrow format, and as such, the resulting data structure could be used by any of the 11 software run-time environments supported by the Arrow project, enabling the use of the accelerator in any of them.

Values Decoder: The internals of the Values Decoder depend on the data types used in the column, since that influences what encoding schemes can be used. Furthermore, the values can be compressed, and therefore must be decompressed for reading.

Decompressor: When compression is used, the Values Converter contains a Decompressor component with a streaming interface. It can be replaced with decompressors for any of the supported Parquet compression schemes, as long as they have a streaming interface, such as e.g. can be found in an open-source implementations of Snappy [32] (performing up to ≈ 8 GB/s) or GZip [33]. For files that are uncompressed, the decompressor may simply pass through the data stream.

Decoders: For decoding, in the prototype implementation of this work, we require that at least primitives (`float`s and `int`s) and variable-length arrays (including UTF8-strings) can be decoded. More elaborate data types could be supported by implementing different decoders, but this is kept for future work.

Parquet allows for various encoding schemes for various data types, of which we will discuss three to meet the aforementioned requirements:

(A) Plain encoding of fixed-size primitives.
(B) Bit-packed delta encoding of fixed-size primitives.
(C) Mixed encoding of UTF8 strings.

In case A, the raw byte representation of the mentioned data types is used. This implementation for primitives requires to simply pass-through the decompressed bytes, but for variable-length arrays it depends on how their size is encoded.

In case B, an initial value is given. Then, for each value, only the difference (delta) with respect to the previous value is stored. When the deviation between values is low, bit-packing results in a small storage footprint for each value, as we can encode the delta with a low number of bits.

In case C, there are several choices. A string can be formatted as a length followed directly by its characters. However, this causes a potential throughput bottleneck, because in this case consecutive strings must be de-interleaved, since the Fletcher Array Writer interface uses a separate length and character stream for the UTF8 string type interface. We found that de-interleaving requires an unacceptably large amount of FPGA resources, because there are many possible combinations of how strings are packed into a single stream word. Each of these combinations would require a separate parallel data path, as the correct output can only be checked after the whole streamed word has been processed.

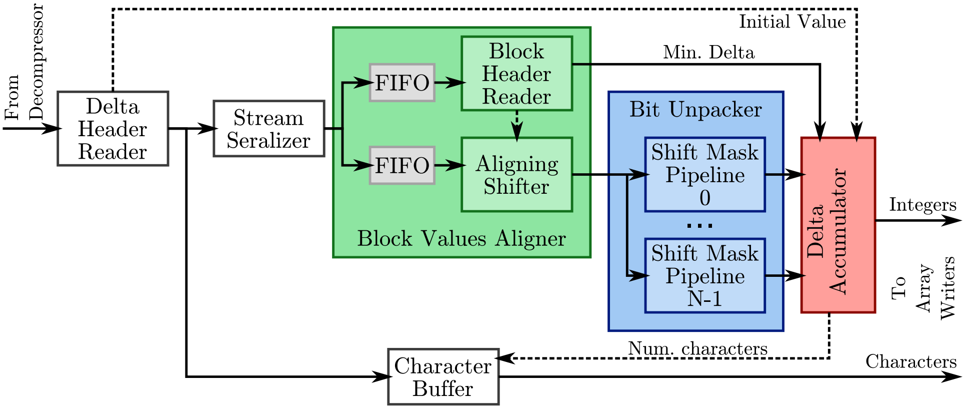


Figure 4.18: Delta decoder

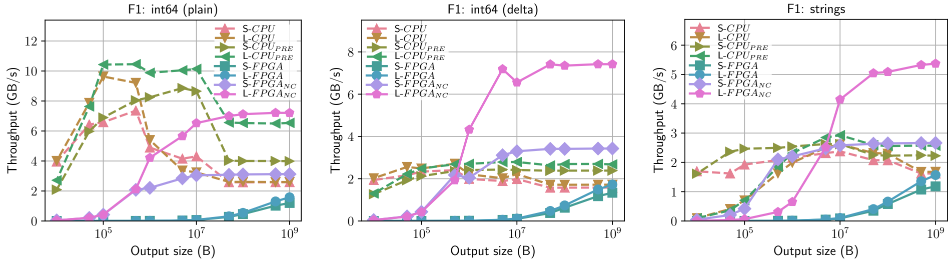
Fortunately, Parquet supports a string format that stores sequences of strings as (bit-packed, delta encoded) lengths and (plain) characters separately within a page. However, the Fletcher Array Writer interface only allows to stream in a single length per cycle[1]. To prevent the Fletcher Array Writer to cause a bottleneck, we have improved the Fletcher DMA engines to work with a parallel prefix sum adder to solve this issue.

Delta Decoder: To decode bit-packed delta-encoded values to raw values (used in aforementioned cases B and C), we implement the *Delta Decoder* component, also shown in Figure 4.18, to be used within the Values Decoder.

It consists of a Delta Header Reader, responsible to read metadata related to the delta encoded values, such as the initial value. It also contains block sizes and number of blocks, since even within the delta encoded values, yet another level of hierarchy exists, that splits delta encoded value runs over multiple blocks. After parsing, the Delta Header Reader aligns input stream to the start of a block. Each block contains more metadata that is parsed by the Block Header Reader; a minimum delta that serves as an offset for the Delta Accumulator. After parsing the Block Header Reader, the stream is again aligned to the first delta encoded value. Through a component called Bit Unpacker, consisting of several shift- and mask pipelines, delivering unpacked deltas, data is finally fed to the Delta Accumulator in parallel. This unit performs the final parallel prefix sum on the initial value, minimum delta and unpacked deltas to obtain the actual values.

4.7.3. RESULTS

We continue to describe measured results on the implementation of the proposed system. We first describe the setup of our experiments, followed by performance measurements and area statistics. We conclude this section with a discussion.

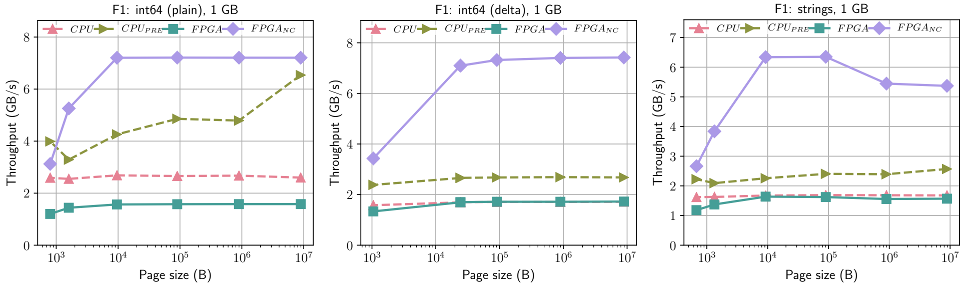


(a) Data: 64-bit ints. Encoding: plain.

(b) Data: 64-bit ints. Encoding: delta.

(c) Data: UTF8 strings. Encoding: mixed.

Figure 4.19: AWS EC2 F1 throughput versus Arrow RecordBatch output size



(a) Data: 64-bit ints. Encoding: plain.

(b) Data: 64-bit ints. Encoding: delta.

(c) Data: UTF8 strings. Encoding: mixed.

Figure 4.20: AWS EC2 F1 throughput versus Parquet page size

EXPERIMENT SETUP

The Parquet-To-Arrow converter is implemented on two platforms; the Amazon EC2 F1 platform using an Intel Xeon E5-2686 v4 CPU and a Xilinx XCVU9P FPGA (hereafter *F1*), and an Inspur FP5290G2 with a dual-socket POWER9 Lagrange 22-core CPU and OpenCAPI interface to an ADM-PCIE-9H7 with a Xilinx XCVU37P (hereafter *OpenCAPI*). The FPGA implementation of the F1 system runs at 250 MHz, while the FPGA implementation of the OpenCAPI system runs at 200 MHz. The implementation is publicly available, free, and open-sourced, including all benchmarks performed to reproduce the result in this section [34].

The FPGA implementation in the F1 system requires the Parquet file to be copied from host memory to on-board memory, because it can only access the on-board DDR memories of the accelerator card. During this transfer, the CPU may perform other tasks, if they can be overlapped. We therefore present two flavors of measurements for the F1 system. In the first, denoted by *FPGA*, we measure the end-to-end solution, where the Parquet file starts in host memory and ends up decoded as an Arrow RecordBatch in host memory again (i.e. a full round-trip). In the second, we measure *no copy* time, denoted by

[1] Because Arrow stores strings as contiguous *offsets* into a character buffer in memory, rather than as contiguous lengths and a character buffer. This makes the offsets buffer a prefix sum of all string lengths, for which Fletcher only calculates one value per cycle.

$FPGA_{NC}$. Here, we measure only the FPGA processing time with the Parquet file already in the on-board DDR memory and the Arrow RecordBatch ending up in the on-board DDR memory as well.

For the FPGA implementation of the OpenCAPI system, we only have one flavor of measurements, denoted by $FPGA$. This includes the whole round trip from host memory, to FPGA, back to host memory. The OpenCAPI system is unique in this sense, since it allows to load and store the data directly from host memory using virtual addresses of the associated process controlling the FPGA accelerator.

To obtain the absolute best result on the CPU, it was necessary to re-implement the Parquet subset supported by the FPGA implementation in `C++` and compile it using `GCC` with `-Ofast`. For the F1 system, `-march=native` was also used, but this flag is not available for the POWER9 CPU of the OpenCAPI system. Existing implementations in Java did not support the Arrow in-memory format yet, and existing implementations in `C++` did not support the latest Parquet specification V2.0 yet, that is used in this work. Our `C++` implementation outperformed the existing software implementations for this subset, providing us with the fastest CPU implementation. We make the same assumptions in all implementations, most importantly that we pre-allocate the resulting Arrow buffers, and do not grow them dynamically, which will involve copy overhead. This software implementation is denoted as CPU.

We create a second `C++` implementation where the virtual memory pages[2] of the Arrow buffers are touched, to make sure the TLB is ‘warm’, consequently removing the overhead of a ‘cold’ TLB from the measurements. This is done to mimic the extreme best case resulting from an often used construct in big data systems; memory pools. These are relatively large and typically zero-initialized virtual memory allocations, that can be rapidly freed when a process exits. Because they are zero-initialized, the pages of the memory pool are already touched, resulting in a ‘warmer’ TLB on average. As such, the best case for this behavior is mimicked, and the most optimistic (albeit less realistic) scenario for the CPU implementations is measured. These measurements are denoted as CPU_{PRE}.

For both the CPU implementation and for the FPGA implementation, we measure the performance of one thread and one kernel, respectively.

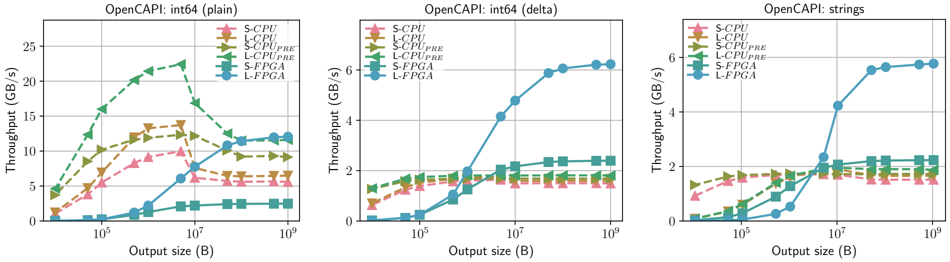
We measure three combinations of data types and encoding, from the cases described in the previous section:

(A) plain encoded 64-bit integers, shown as *int64 (plain)*
(B) bit-packed delta encoded 64-bit integers, shown as *int64 (delta)*
(C) UTF8 strings where the lengths use delta encoding, shown as *strings*

Data for *int64 (plain)* was randomly generated. Data for *int64 (delta)* was randomly generated, but with a random modulo that changes every 256 elements. This modulo creates a mix of data requiring different packing lengths, instead of almost always requiring the full width as is expected for fully random data. The emphstrings were randomly generated with random lengths between 1 and 12 characters. Note that the length of the strings determines the mix between delta-packed and plain data, and very long strings will result in behavior more similar to that of *int (plain)*.

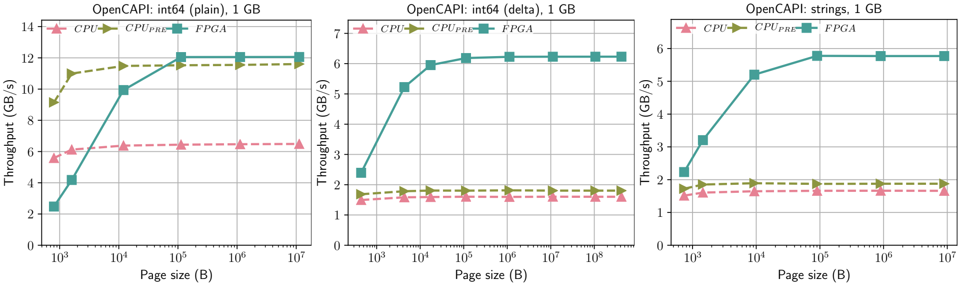
[2] Not to be confused with Parquet pages.

PERFORMANCE



(a) Data: 64-bit `int`s. Encoding: plain. (b) Data: 64-bit `int`s. Encoding: delta. (c) Data: UTF8 strings. Encoding: mixed.

Figure 4.21: POWER9/OpenCAPI/9H7 throughput versus Arrow RecordBatch output size



(a) Data: 64-bit `int`s. Encoding: plain. (b) Data: 64-bit `int`s. Encoding: delta. (c) Data: UTF8 strings. Encoding: mixed.

Figure 4.22: POWER9/OpenCAPI/9H7 throughput versus Parquet page size

AWS EC2 F1 - Throughput vs. RecordBatch size For the F1 system, we first measure the throughput versus the size of the resulting Arrow RecordBatch, shown in Figure 4.19. The figures display two Parquet page sizes; small pages, prefixed with "S-", where the pages are approximately 1 kB in size, and large pages, prefixed with "L-", where the pages sizes are approximately 10 MB in size.

Since plain encoded 64-bit integers require no further decoding, they correspond to performing a plain copy. Figure 4.19a therefore gives a good indication of the overhead associated with processing the Parquet files on FPGA. When the output size is very small (i.e. small Arrow RecordBatches), the overhead of initializing the FPGA to start operating becomes evident. As the output size grows, we see that the FPGA accelerated solution increases in bandwidth, since it has to spend relatively less time on initialization. Looking at the $FPGA_{NC}$ curve, we furthermore observe that the proposed system can saturate the on-board bandwidth of one of the DDR controllers of the F1 accelerator card. The read and write interface use the same DDR controller. Since the interface operates in half-duplex at 16 GB/s, and since we both read and write at the same time, the throughput saturates just over 7 GB/s. The accelerator for this configuration does not outperform the

CPU implementations because the round-trip throughput does not reach above 2 GB/s. While this configuration is not typical for Parquet files (typically both delta encoding and compression would be applied), it does reveal the overhead associated with initializing the FPGA solution quite well. The CPU solutions initially increase in throughput, but later decrease around the tens of megabytes range, most likely due to running out of cache space to store the Arrow RecordBatch.

In the delta-encoded and bit-packed integer case (*int64 (delta)*), shown in Figure 4.19b, there is some actual computation to perform next to decoding the page headers. In this case, the integers also need to be unpacked, better revealing the value of the proposed system. Now that calculations have to be performed, we quickly find the $FPGA_{NC}$ to outperform the CPU implementation in terms of processing throughput, achieving close to the available DDR interface bandwidth again. However, the end-to-end measurement ($FPGA$) reveals that the overhead associated with making copies from and to the on-board DDR memory still prevents the FPGA accelerated solution from achieving better performance.

Finally, in the *strings* case, shown in Figure 4.19c, the same conclusions as for the *int64 (delta)* case can be drawn.

AWS EC2 F1 - Throughput vs. Parquet page size Also for the F1 system, we measure the throughput versus various Parquet page sizes, shown in Figure 4.20. For all measurements in this figure, an output RecordBatch of 1 GB in size was chosen.

For all data types, around the default Parquet page size of 1 MiB, we observe that the available DDR controller bandwidth can already be saturated. When the page sizes are set to be much smaller, we observe that the overhead of decoding the pages becomes a bottleneck. Only for the plain encoded integers, the processing throughput drops below that of the CPU implementation, but for the other data types and encoding, even for non-realistic page sizes of around one kB, the $FPGA_{NC}$ measurement throughput is better. The curve for the $FPGA$ measurement is lower for plain encoding, but roughly the same as the CPU curve for the other data types, but still not better than the CPU_{PRE} implementation.

POWER9/OpenCAPI/9H7 - Throughput vs. RecordBatch size For the OpenCAPI system, we also measure the throughput versus the size of the resulting Arrow RecordBatch, shown in Figure 4.21. We do this in the same way as for the F1 system, using very small (prefixed with "S-" at approximately 1 kB) and large pages (prefixed with "L-" at approximately 10 MB).

For all data types and encodings, the FPGA implementation is able to outperform the CPU implementation when the page sizes are sufficiently large and the output size is large. For the plain encoded integers, the benefit is rather small, but for configurations where actual work has to be performed to decode the data, the FPGA implementation shows its value, resulting in a speedup of around 3×. For the delta encoded integers and the strings, the throughput of the FPGA implementation levels off at around 6 GB/s, while the CPU reaches 1.5 – 2 GB/s. This is not due to the OpenCAPI interface, that allows bandwidths of up to 25 GB/s, but due to the delta decoding step. The difference in throughput compared

Data type	**Enco-ding**	**Input stream width** (bits)	**LUTs** (%)	**Regis-ters** (%)	**BRAM** (%)
Int64	Plain	512	1.18	1.27	2.13
Int32	Delta	64	1.46	1.50	2.85
Int32	Delta	128	1.55	1.61	2.99
Int64	Delta	64	1.68	1.64	2.66
Int64	Delta	128	1.76	1.76	2.99
Int64	Delta	256	1.90	1.99	3.24
UTF8	Mixed	128	2.79	2.92	4.47

Table 4.8: Resource utilization. Device: Xilinx XCVU9P.

to the $FPGA_{NC}$ measurement of the F1 system is explained by the difference in clock frequency.

POWER9/OpenCAPI/9H7 - Throughput vs. Parquet page size We also measure the throughput versus various Parquet page sizes for the OpenCAPI system, shown in Figure 4.22. Around the default Parquet page size of 1 MiB, the FPGA throughput is already saturated. Similar as in the discussion of the F1 system, the measurements of the smallest (although rather unrealistic) page sizes reveal the overhead associated in decoding pages.

RESOURCE UTILIZATION

In Table 4.8, we find the resource utilization statistics of a single Parquet-to-Arrow converter for the Xilinx XCVU9P. For clarity, this excludes the F1 platform-specific resources. The area utilization is modest, with most resources staying under 5%. This allows for multiple converter cores to be implemented in contemporary FPGAs, that could work on converting the Parquet file in parallel, leveraging its parallel-friendly format. Timing closure for all designs was reached for a 250 MHz clock rate.

DISCUSSION

From the results presented, we find the Parquet-to-Arrow converter accelerator to be an interesting alternative to a CPU-only solution. We stipulate the following general observations.

First, our measurements indicate that CPUs will become the bottleneck when loading data from the Parquet file format, rather than I/O bandwidth, for modern storage systems with increased I/O bandwidth. This is most pronounced by first looking at the difference between the CPU measurements of plain encoded integers in Figures 4.19a and 4.21a. Since the operation to decode the Parquet file only requires parsing page headers and otherwise only performing `memcpy`, the CPUs are bottlenecked by memory I/O bandwidth, achieving close to 7 GB/s and 12 GB/s when the output size is larger than the caches. However, continuing to look at Figures 4.19b,4.19c and Figures 4.21b,4.21c, where actual work on decoding has to take place, the CPU performance never reaches above 3 GB/s anymore. The CPU has become the bottleneck.

Second, the main figure of merit being throughput, the FPGA accelerator always outperforms the CPU implementations in terms of processing throughput. To increase the end-to-end bandwidth of the F1 system, it would be interesting to explore overlapping data copy and FPGA computation, where a single instance of our proposed architecture is already able to saturate a PCIe interface.

The OpenCAPI interface provides up to 25 GB/s of full-duplex bandwidth. Reaching this bandwidth is not easy to achieve with a single kernel, since closing timing for the delta decoding step is too hard when the interface is very wide. However, since various parts of a Parquet file may be decoded in parallel, multiple instances of our proposed design would be able to saturate the interface bandwidth with ease. Such a design is very feasible since the area footprint is relatively small and could by estimation fit more than sixteen times in the VU37P, where four instances should saturate the bandwidth in practice.

Finally, as the Parquet format is of a very dynamic nature, it is challenging to support all possible potential configurations. A Parquet file can only be converted by the accelerator if the facilitated configurations of data type, decompressor and decoder match the columns of interest in the file. If switching between different configurations is required, it is required to extend the decoding components in such a way that the area overhead may be rather large. It would be more beneficial to maintain a library of pre-synthesized configurations that can be (partially) reconfigured into the converter depending on the file. This would leverage the reconfigurability advantage of the FPGA while allowing a Parquet-To-Arrow converter to maintain a relatively small footprint. At the same time, performance implications of such a mechanism should first be evaluated, especially when files are small, since in that case, the reconfiguration overhead may dominate.

4.7.4. CONCLUSION

As I/O bandwidth of storage and network continues to increase, the use of traditional exchange formats for large data structures leaning on the premise of fast CPUs and slow I/O will begin to see CPU bottlenecks. We observed that this bottleneck is also present when loading data from Apache Parquet files into Apache Arrow in-memory data structures at I/O bandwidths of modern storage and network solutions, where the CPUs of our systems are only able to support a throughput in the order of several GB/s.

We proposed an FPGA accelerator design, in which the Parquet files are converted to Apache in-memory data structures, only using the CPU to parse high-level metadata that does not impact performance. We continued to provide a modular and extensible architecture that is able to parse lower-level but more performance-critical metadata about Parquet pages, in addition to being able to decompress and further decode stored values. The architecture is designed in a modular way, allowing users to insert their own decompressors and decoders, based on the many possible encodings that Parquet files may employ. We present an implementation for three data types and encodings in this paper, for plain encoding, and for bit-packed delta-encoded values for both integers and UTF8 strings.

When using encoding schemes more complex than *plain*, results clearly show the merit of using FPGAs. For a POWER9 system with an FPGA connected via OpenCAPI, a single instance of the proposed architecture is able to process realistically configured Parquet files at up to 6 GB/s versus just over 2 GB/s for an optimized CPU implementation. On an Amazon EC2 F1 system, similar advantages are measured for cases where data transfer between host and FPGA can be overlapped. Otherwise, interface bandwidth becomes a bottleneck and performance does not surpass the CPU. The implementations use a small amount of resources (below 5%), which allows multiple instance of the Parquet-To-Arrow converter to be instantiated. This will allow processing Parquet pages in parallel, increasing throughput as long as there are resources and I/O bandwidth available.

When compression schemes such as GZip, Snappy or Brotli would be applied on top of the presented delta encoding, the FPGA accelerator benefits are expected to become more pronounced. These operations will only decrease CPU performance, while there are high-throughput, fully streamable implementations with low resource utilization available for FPGAs. Integration and evaluation of such designs are envisioned for future work. In conclusion, the Parquet-To-Arrow converter is a promising heterogeneous alternative to CPU-only based processing of Parquet files into Arrow in-memory data structures.

4.8. CONCLUSION

In this chapter, we have demonstrated six applications with the help of (parts of) the features of Fletcher described in the previous chapter. The first three applications, regular expression matching, k-means clustering, and writing strings to memory at high bandwidth, specifically focus on the increased performance when serialization overhead is prevented through the use of Arrow. We briefly explored integrating with a commercial HLS tool. Three use-cases show that the combination of Arrow and Fletcher can be beneficial to the end-to-end throughput of an FPGA accelerated application, especially when the accelerated operation is streamable. For these cases, the benefit was shown to range from 1.3× - 49×, depending on the characteristics of the applications and the implementation platform. For a fourth use case that uses an HLS-based design flow, Fletcher allows the kernel to be expressed using stream arguments rather than buffer pointer arguments, increasing the ease of use and integrated performance of a commercial HLS tool.

We have described the acceleration of Variant Calling in genomics, more specifically we analyzed the efficiency of systolic arrays that implement the PairHMM Forward Algorithm to find the overall alignment probability of a read to a haplotype. We have shown architectures that can implement fixed-size systolic arrays in such a way that on average around 90% of the circuit is effectively used when processing a real dataset, We have described the Arrow Schema that may be used to represent the structured data, and have upgraded the design to make use of Fletcher's Array Readers and Writers and posit arithmetic, increasing the benefit over CPU computations by more than three orders of magnitude, as posit arithmetic can only be emulated in software. This shows furthermore the short-term specialization advantage of FPGA accelerators.

Finally, as I/O bandwidth of storage and network continue to increase, the use of traditional exchange formats for large data structures leaning on the premise of fast CPUs and slow I/O will begin to see CPU bottlenecks. We observed that this bottleneck is also present when loading data from Apache Parquet files into Apache Arrow in-memory data structures at I/O bandwidths of modern storage and network solutions, where the CPUs of our systems are only able to support a throughput in the order of several GB/s. We proposed an FPGA accelerator design using Fletcher, in which the Parquet files are converted to Apache in-memory data structures, only using the CPU to parse high-level metadata that does not impact performance. For a POWER9 system with an

FPGA connected via OpenCAPI, a single instance of the proposed architecture is able to process realistically configured Parquet files at up to 6 GB/s versus just over 2 GB/s for an optimized CPU implementation.

REFERENCES

[1] Google, *RE2, a regular expression library,* (2018).

[2] J. van Straten, *vhdre: a vhdl regex matcher generator,* (2019).

[3] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, *Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow,* in *Applied Reconfigurable Computing,* edited by C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz (Springer International Publishing, Cham, 2019) pp. 32–47.

[4] J. Shendure and H. Ji, *Next-generation dna sequencing,* Nature biotechnology **26**, 1135 (2008).

[5] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids,* 1st ed. (Cambridge University Press, 1998).

[6] M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. Del Angel, M. A. Rivas, M. Hanna, *et al.*, *A framework for variation discovery and genotyping using next-generation dna sequencing data,* Nature genetics **43**, 491 (2011).

[7] A. C. Jacob, J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, *Preliminary results in accelerating profile HMM search on FPGAs,* in *2007 IEEE International Parallel and Distributed Processing Symposium* (IEEE, 2007) pp. 1–8.

[8] K. Benkrid, P. Velentzas, and S. Kasap, *A high performance reconfigurable core for motif searching using profile HMM,* in *Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on* (IEEE, 2008) pp. 285–292.

[9] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu, *Accelerating HMMer on FPGAs using systolic array based architecture,* in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (IEEE, 2009) pp. 1–8.

[10] M. N. M. Isa, K. Benkrid, and T. Clayton, *A novel efficient FPGA architecture for HMMER acceleration,* in *2012 International Conference on Reconfigurable Computing and FPGAs* (IEEE, 2012) pp. 1–6.

[11] S. Ren, V.-M. Sima, and Z. Al-Ars, *FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis,* in *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (IEEE, 2015) pp. 1465–1470.

[12] M. Ito and M. Ohara, *A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm,* in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)* (IEEE, 2016) pp. 1–3.

[13] M. Brobbel, *CAPI Streaming Framework,* https://github.com/mbrobbel/capi-streaming-framework (2016).

[14] J. Peltenburg, S. Ren, and Z. Al-Ars, *Maximizing systolic array efficiency to accelerate the pairhmm forward algorithm,* in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2016) pp. 758–762.

[15] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, *et al.*, *The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data,* Genome research (2010).

[16] L. van Dam, J. Peltenburg, Z. Al-Ars, and H. P. Hofstee, *An accelerator for posit arithmetic targeting posit level 1 blas routines and pair-hmm,* in *Proceedings of the Conference for Next Generation Arithmetic 2019*, CoNGA'19 (Association for Computing Machinery, New York, NY, USA, 2019).

[17] J. L. Gustafson, *The End of Error: Unum Computing* (CRC Press, 2015).

[18] Boost, *cpp_dec_float - 1.63.0,* http://www.boost.org/doc/libs/1_63_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/cpp_dec_float.html (2013), [Online; accessed 2018-11-20].

[19] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield, *Non-volatile storage,* Queue **13**, 20:33 (2015).

[20] F. Kruger, *Cpu bandwidth – the worrisome 2020 trend,* (2016).

[21] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, *Performance analysis of nvme ssds and their implication on real world databases,* in *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15 (Association for Computing Machinery, New York, NY, USA, 2015).

[22] B. Kim, J. Kim, and S. H. Noh, *Managing array of ssds when the storage device is no longer the performance bottleneck,* in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (USENIX Association, Santa Clara, CA, 2017).

[23] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler, *Albis: High-performance file format for big data systems,* in *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (USENIX Association, Boston, MA, 2018) pp. 615–630.

[24] M. Ajdari, P. Park, J. Kim, D. Kwon, and J. Kim, *Cidr: A cost-effective in-line data reduction system for terabit-per-second scale ssd arrays,* in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2019) pp. 28–41.

[25] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, *The end of slow networks: It's time for a redesign,* Proc. VLDB Endow. **9**, 528–539 (2016).

[26] L. Kuhring, E. Garcia, and Z. István, *Specialize in moderation—building application-aware storage services using fpgas in the datacenter,* in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)* (USENIX Association, Renton, WA, 2019).

[27] The Apache Software Foundation, *Apache Parquet,* (2018).

[28] The Apache Software Foundation, *Apache Arrow,* (2018).

[29] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee, *Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow,* in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (2019) pp. 270–277.

[30] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, *Dremel: Interactive analysis of web-scale datasets,* in *Proc. of the 36th Int'l Conf on Very Large Data Bases* (2010) pp. 330–339.

[31] Google, *Snappy,* .

[32] Delft University of Technology, *Hardware snappy decompressor,* .

[33] Xilinx, *Vitis gzip implementation,* .

[34] L. van Leeuwen and Accelerated Big Data System Group, Delft University of Technology, *fast-p2a: An FPGA-based Parquet-to-Arrow converter,* .

[35] *Battling the cpu bottleneck in apache parquet to arrow conversion using fpga (under review),* (2020).

[36] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, and et al., *The FitOptiVis ECSEL Project: Highly Efficient Distributed Embedded Image/Video Processing in Cyber-Physical Systems,* in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19 (Association for Computing Machinery, New York, NY, USA, 2019) p. 333–338.

PERSONAL CONTRIBUTIONS, ACKNOWLEDGEMENTS AND FUNDING.
This chapter is based on the three published articles [29][14][16] and one article in preprint at the time of writing [35]. L. Wijtemans designed the k-means clustering application and performed the experiments on Amazon EC2 F1. L. van Leeuwen designed and implemented the Parquet-to-Arrow converter and performed the experiments on Amazon EC2 F1. J. Hoozemans ported the Parquet-to-Arrow converter and experiments to the POWER9/OpenCAPI platform. J. van Straten designed the *vhdre* tool used to generate the hardware for the regular expression matchers. S. Ren provided background information and data sets for the PairHMM accelerator experiments. L. van Dam upgraded the PairHMM design with posits and Fletcher, performed the experiments, and wrote the posits paper. J.W. Peltenburg designed all other accelerators and experiments, wrote the

Fletcher and Parquet papers, designed the initial version of the PairHMM accelerator, and partially advised the work of Wijtemans, Van Leeuwen, and Van Dam.

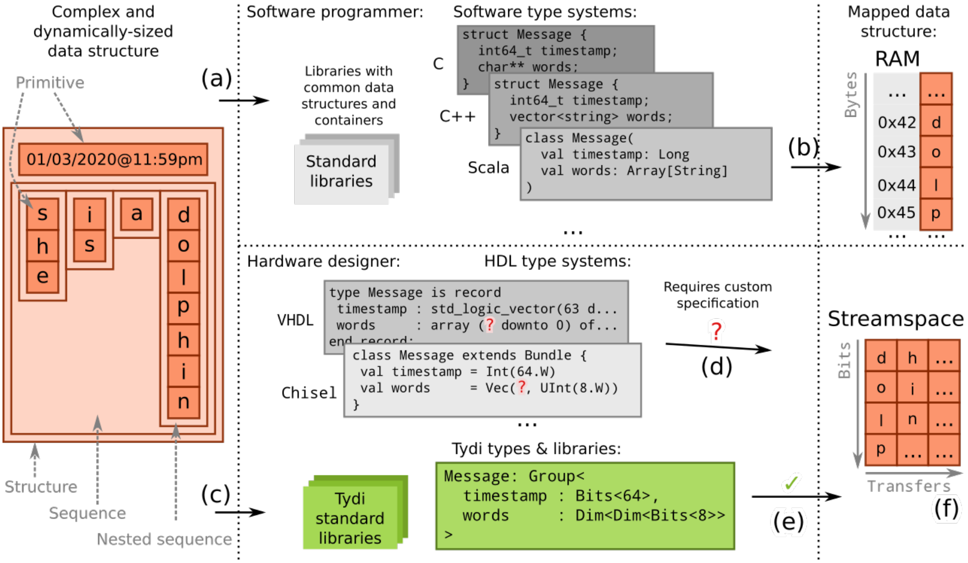
The authors would like to thank IBM (in particular Jinho Lee for helping us run many experiments), Xilinx and AlphaData for their support.

This work was partially supported by the European Commission in the context of the ARTEMIS project ALMARVI (project #621439). This work was partially funded by Xilinx. Some of this work is part of the FitOptiVis project [36], funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162.

5

Complex data structures over hardware streams

Based on the lessons learned from the development of the Fletcher framework, this chapter introduces a type system and streaming interface specification that help to define how complex, nested, and dynamically sized data structures can be exchanged between components in digital circuits. We believe this type system and specification, called Tydi, are helpful in the future development of new hardware generation tools such as Fletcher itself, but also for new hardware description languages, providing a high level of abstraction without a loss of performance, while still staying hardware-oriented. Through the simple but intuitive type system, it is possible to construct types for interfaces over which all sorts of data structures may be transferred, and through the specification, the associated access behavior is clearly defined. We provide an initial set of standard generic types for common data structures, that serve as an analogy for containers in standard libraries of modern software languages. Such containers are well known by software developers, and they select them according to the access behaviour of their applications and the implications of that access behaviour for that container. However, hardware streams deliver data structures in an inherently sequential manner, while software abstractions typically assume random-access, a difference that is reflected in the constructs of the type system and the containers that we provide. Despite the abstractions that Tydi provides, at the digital-circuit level, developers may still make hardware-oriented trade-offs in area and throughput when necessary. We implement a code generation tool that can target various contemporary hardware description languages to generate boilerplate code for interfaces based on Tydi types. A brief study of the amount of code to define a Tydi type and the amount of generated boilerplate code shows that the abstractions provided by Tydi may decrease developer effort by orders of magnitude.



```
struct Message {
  int64_t timestamp;
  char** words;
}
```

```
struct Message {
  int64_t timestamp;
  vector<string> words;
}
```

```
class Message(
  val timestamp: Long
  val words: Array[String]
)
```

```
type Message is record
 timestamp : std_logic_vector(63 d...
 words     : array (? downto 0) of...
end record;
```

```
class Message extends Bundle {
 val timestamp = Int(64.W)
 val words     = Vec(?, UInt(8.W))
}
```

```
Message: Group<
  timestamp : Bits<64>,
  words     : Dim<Dim<Bits<8>>
>
```

Figure 5.1: Tydi context.

5.1. INTRODUCTION

Exchanging data between components of a computing system is a major topic in computer architecture. When components interact, a well-specified representation of the data should exist in whatever medium used for communication to allow the data to be interpreted correctly and enable reusable and extensible designs. Clear format specifications are especially useful for an open-source community, where it enables more efficient collaboration.

Agile development of hardware-oriented solutions is driven by many excellent open-source projects that increase the level of abstraction at which hardware is described. Some experts even argue that we are in “A Golden Age of Hardware Description Languages” [1] — more advanced designs can be automatically synthesized from fewer lines of code.

However, we observe a lack of standardized exchange formats and abstract views for complex data structures at the level of digital circuits. As a result, developers often manually design their custom representations of more advanced composite and aggregate data structures (e.g. strings, nested lists, etc.), that need to be exchanged between components over streams.

We propose **Tydi**; an open specification (found freely online [2]) that allows developers to map composite and dynamically-sized data structures onto hardware streams. It furthermore provides an abstract, but still hardware-oriented view of these data structures, as to not lose the opportunity to make common trade-offs in the design phase.

An overview of the context of Tydi is seen in Figure 5.1. To implement a data structure, programmers choose some types and containers, helped by language constructs and libraries (a), run-time engines and compilers take care of the mapping to RAM (b). The same for contemporary HDLs (c) is prevented because dynamically-sized structures are

Data type	**Data structure (or instance)**	**Description**
$\textsc{Empty}$	$(\emptyset)$	Empty set, singleton value.
$\textsc{Prim}\langle B\rangle$	$(b_{B-1}, b_{B-2}, ..., b_0)$	Primitive element containing B bits of information.
$\textsc{Struct}$ $\langle T_1, T_2, ..., T_n\rangle$	$(\mathrm{I}(T_1, p_1), \mathrm{I}(T_2, p_2), ..., \mathrm{I}(T_n, p_3))$	Composite type. An instance is a set with one instance of every type argument $T_1, T_2, ..., T_n$.
$\textsc{Variant}$ $\langle T_1, T_2, ..., T_n\rangle$	$\mathrm{I}(t \in (T_1, T_2, ..., T_n), p_t)$	A variant type. An instance is one of either type T_1 or T_2, etc. t is known when instantiated, by some tag.
$\textsc{Tup}\langle n, T\rangle$	$(\mathrm{I}(T, p_1), \mathrm{I}(T, p_2), ..., \mathrm{I}(T, p_n))$	A fixed-length aggregate type. An instance is a sequence with $n \in \mathbb{N}^+$ instances of the same type T. n is part of the type.
$\textsc{Seq}\langle T\rangle$	$(\mathrm{I}(T, p_1), \mathrm{I}(T, p_2), ..., \mathrm{I}(T, p_n))$	A variable-length aggregate type. An instance is a sequence with $n \in \mathbb{N}^0$ instances of the same T. n is only known when instantiated.

$\mathrm{I}(T, p)$ is an instance of type T, where p parametrizes the instance, if necessary.

Table 5.1: Conceptual view of data types and data structures used throughout this article.

not inherently supported. When mapping to hardware streams (d) designers customize solutions to transport data structures over multiple stream transfers. Tydi is a specification that clearly and intuitively provides a mapping (e) and pre-defined containers for common types.

At the core of the specification lies a type system. It provides an intuitive and clear definition of how complex data structures are transported over hardware streams. We discuss additional parameters that can be used to make an area/throughput trade-off for component interfaces, and provide a precise specification at the hardware level. This specification can be used by developers that design components, or that combine components into larger designs, either manually or by automated tools.

5.2. BACKGROUND

Designing digital circuits from a dataflow-oriented perspective involves selecting appropriate transformations and connections between the transformations through directed channels. When data starts flowing from external sources, the specific configuration of transformations and channels allow an algorithm to be executed, producing output that can flow back to an external sink. For digital circuits, channels are often implemented as *streams*; point-to-point connections, where a sink receives data elements from a source in FIFO-order. Transformations are typically implemented as **streamlets**: components with streaming interfaces.

When data structures flow over streams between streamlets, it is favorable to reason about them at a high level of abstraction, rather than at a low level (bits and clocks), especially when the data structures are dynamic and complex. An example data structure that we will use throughout this article, is a chat message consisting of a (64-bit POSIX-time encoded) timestamp and a sentence (Extended-ASCII encoded string). To create more context, envision an application with an unbounded stream of messages, where one would like to apply a transformation that filters the message by some time range, then splits the sentence into separate words.

A more formal view of data types and structures is presented in Table 5.1. Using that view, we can describe the aforementioned chat message as: $T_m = \textsc{Struct}\langle\textsc{Prim}\langle 64\rangle, \textsc{Seq}\langle\textsc{Prim}\langle 8\rangle\rangle\rangle$, and the filtered message as: $T_f = \textsc{Struct}\langle\textsc{Prim}\langle 64\rangle, \textsc{Seq}\langle\textsc{Seq}\langle\textsc{Prim}\langle 8\rangle\rangle\rangle\rangle$.

In the software domain, instantiated data structures of these types are typically materialized as bytes in a RAM. How this is done depends on the software framework used, as shown in Figure 5.1. The exact byte-level representation is left to compilers, run-time engines and standard libraries. Especially for aggregate types, programmers typically select pre-defined *containers* from standard libraries (e.g. the C++ `std::vector`) that help mapping tuples and sequences based on their basic notion of the architecture of their device (typically a load-store architecture) and properties of their algorithm/workload (e.g. whether to store a sequence as a linked-list or in a hash-table). This greatly abstracts the details of *how* the data structure is mapped onto (typically) a RAM — a one-dimensional sequence of bytes, under some constraints (the total number of bytes available), but programmers retain some control over the performance characteristics of the mapping.

While attempting to map complex and dynamically-sized data structures onto a single streamed element, one quickly finds it impractical to allow the streamed element to be as wide as the amount of information in bits. This impracticality exists for at least two reasons. First, some aggregate types, such as the sequence, are dynamically-sized. Accommodating an interface at design-time based on some initial guess for its length would rule out support for potentially larger sequences. Second, data structures described by aggregate types that *are* statically-sized, such as tuples, can grow arbitrarily large. Streamlets may not be able to absorb all data from a large element at once. Consequently, one would be under-utilizing the resources used for the streaming interface.

Thus, designers often choose to split the information over multiple stream transfers, such that over time, the whole data structure is transported between the sourcing and sinking streamlet. Therefore, a hardware developer does not map a data structure merely onto space (e.g. a one-dimensional bit-vector), but also onto time, or more specifically, stream transfers. From this description, a two-dimensional plane emerges that we will call **streamspace** — the plane consisting of both a spatial resource (bits) and a resource of temporal nature (transfers).

To the best of our knowledge, while there is an enormous body of work in the software domain about mapping complex data structures onto byte-addressed RAM, little literature exists that discusses methods of mapping composite, potentially dynamically-sized and nested aggregate types onto streamspace from an abstract point of view. This causes the tedious need for hardware designers to create custom formats for their designs and data structures (often on top of existing standards), which is a problem we address through Tydi.

5.2.1. Related work

One widely-used streaming protocol specification is the AXI4-Stream protocol [3]. Users can transport anywhere between zero and N bytes per transfer, with an (optional) `last` bit that denotes the end of a one-dimensional sequence of bytes. It therefore specifies how to transport either Prim⟨8⟩ or Seq⟨Prim⟨8⟩⟩. It does not specify how structures that are not byte-oriented or that have deeper levels of nesting, e.g. Seq⟨Seq⟨Prim⟨7⟩⟩⟩, should be communicated. Avalon Streaming [4] is similar to AXI, but slightly less restrictive, because elements can be arbitrarily sized.

CoRAM++ [5], where DMA engines are generated based on a set of specific C-style data structures, such as multidimensional arrays, linked lists, and trees, allows streamlets to interact with more advanced data structures in memory, but does not focus on communication between streamlets or on how to mix the above data structures.

We have explored active (open-source) hardware frameworks, including classical HDLs (VHDL, Verilog, SystemVerilog) and contemporary ones (Cλash [6], Chisel [7], and Spatial [8]). All these HDLs support compound types that map onto bit-vectors (e.g. VHDL's `record`, Chisel's `Bundle`, etc.), and statically-sized aggregate types, but lack inherit support for dynamically-sized aggregate types mapped onto streamspace. This is unsurprising; the type systems of these frameworks reason only about space, but not about stream transfers — the latter being typically left to the designer — as the goal is to describe hardware just above the register-transfer level. In libraries of some of the languages, abstractions for streaming dataflow designs are provided, e.g. Chisel's `DecoupledIO`, Spatial's `StreamIn/Out` and Cλash's `DataFlow`. The abstractions move towards the level we envision when composing designs out of streams and streamlets, but only abstract the handshake mechanism for otherwise completely user-defined signals, lacking inherent support for throughput scaling of streams that is available in AXI/Avalon.

Commercial high-level-synthesis frameworks (including Vivado HLS and SDAccel) support streams as parameters for functions, creating a streaming interface for kernels. These streams provide an abstraction for the handshake protocol of a single unbounded stream for statically-sized composite types. Information about the size of dynamically-sized aggregate types traveling over the stream still requires a custom mapping onto the streamed elements.

5.3. ENTERING STREAMSPACE

As mentioned in the previous sections, our goal is to find a suitable mapping of the data structures shown in Table 5.1 into streamspace. We propose a mapping, where we define **logical streams**; streams that transport a top-level data structure (that may consist of nested data structures). Depending on the data structure, a logical stream can consist of multiple **physical streams**; streams with their own handshake/transfer interface.

To facilitate a clear definition of the physical streams emerging from a logical stream, we introduce a *streamspace*-oriented type system. The type system exposes the direction of physical streams, and how two of their properties E and D are derived. E is the number of bits of an *element* that the stream transports in every transfer, and D is the number of bits used to signal the end of some (nested) sequence. The physical streams have more properties that are explained in the next section.

At least three use-cases for this type system exist. First, it can be used in tools that automatically generate streamlet interfaces for traditional hardware description languages (e.g. VHDL or (System)Verilog). In a later section, we briefly discuss two implementations of such generators; the reference implementation utility of Tydi, and Fletcher, a hardware acceleration framework for FPGAs. Second, the type system can be used in hardware description frameworks, such as Chisel. Chisel has highly generative capabilities through its host language Scala. The type system and generative code can reside in a Scala library. Third, we envision tight integration within hardware description languages that use a functional programming paradigm, such as Cλash, as they are highly suitable to express dataflow designs.

Type	Description	D_{child}
Bits$\langle B\rangle$	Defines a B-bits primitive element, where $B \in \mathbb{N}_0$.	n/a
Group$\langle S_1, S_2, ..., S_n\rangle$	Concatenates elements of types $S_1, S_2, ..., S_n$ into one physical stream element.	D_p
Union$\langle S_1, S_2, ..., S_n\rangle$	Defines a B-bits element, where B is the max. element width of $S_1, ..., S_N$	D_p
Dim$\langle S\rangle$	Creates a streamspace of elements of type S in the next **dimension** w.r.t. its parent.	$D_p + 1$
Rev$\langle S\rangle$	Creates a new physical stream of S that flows in **reverse** direction w.r.t. its parent.	D_p
New$\langle S\rangle$	Creates a **new** physical stream in the parent space D_p with elements of type S.	D_p

D_0 is the first streamspace dimension, D_p is the dimension of the parent type, if applicable.

Table 5.2: Overview of streamspace types in Tydi

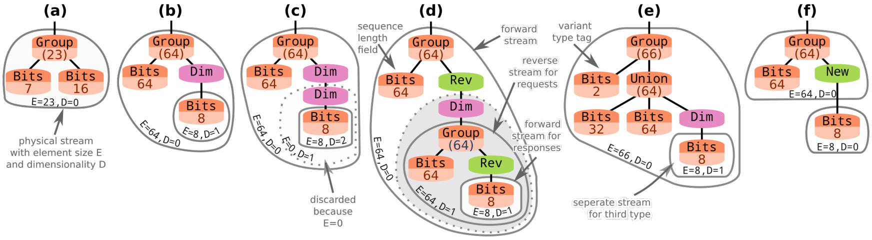


Figure 5.2: Examples of streamspace types.

5.3.1. A stream-oriented type system

We define six types that help to construct a streamspace representation of the data structures, also shown in Table 5.2. These types abstract indivisible properties of data structures being exchanged in streamspace. More advanced abstractions can be constructed by combining these types, as shown in Figure 5.2 (discussed later).

The first three types in the table manipulate the size E of the element that a physical stream transports. As such, they could 'live' outside streamspace (i.e. they map only to a one-dimensional bit vector). The other types are used to create separate physical streams in streamspace.

Of the element-manipulating types, the first, Bits$\langle B\rangle$ will add B bits to that element, and could be seen as simply adding a field of a primitive type to the streamed element. This is the streamspace representation of a Prim$\langle B\rangle$. The second, Group$\langle S_1, S_2, ..., S_n\rangle$, concatenates elements of its child types (where S denotes a streamspace type parameter). This causes the element size E to be the sum of all child element sizes, as long as these children reside in the same physical stream. Group therefore allows to represent Struct, but can also help to combine multiple physical streams, as the type arguments are not limited to element-manipulating types. The final element-manipulating type is Union$\langle S_1, S_2, ..., S_n\rangle$, that selects the element size to be the largest element size of its children. This is useful in representing the Variant type.

Of the physical stream creating types, Dim$\langle S\rangle$ increases the *dimensionality* of its child type S, and therefore increases the parameter D. In physical streams, D bits are reserved that signal an element is the *last* element in a (nested) sequence (rather than e.g. the single 'last' bit of AXI4-Stream). A separate physical stream is created over which zero or more instances travel for every *single* element of its parent. This makes Dim⟨S⟩ suitable to represent (nested) sequences. Rev⟨S⟩ is used to create a physical stream that flows in the reverse direction respective to its parent. This stream remains in the same dimension as its parent; for every element that the parent transfers, also one instance of Rev⟨S⟩ will be transferred. Rev⟨S⟩ can be used for interfaces between streamlets that work on a request-response basis.

New⟨S⟩ is used to create a new physical stream that has the same dimensionality as its parent, and is implicitly at the root of all streamspace types.

In Figure 5.2, we demonstrate by example how data structures can be mapped into streamspace.

(a) A streamspace mapping of a structure with a seven-bit field and a sixteen-bit field: Struct⟨Prim⟨7⟩, Prim⟨16⟩⟩. In the mapping Group⟨Bits⟨7⟩, Bits⟨16⟩⟩, Group concatenates the Bits elements together into a single element, resulting in a single physical stream transporting twenty-three-bit elements ($E = 23$) with dimensionality $D = 0$.

(b) The type T_m of our chat message example. A simple mapping of T_m into streamspace is: Group⟨Bits⟨64⟩, Dim⟨Bits⟨8⟩⟩⟩ creating two physical streams; one for the timestamp field, and another, logically nested in the first, for the sequence of 8-bit elements. For every transfer on the first stream, there must be at least one (possibly empty) transfer on the second stream.

(c) Output T_f of the streamlet transforming T_m. The second field is now a *sequence of sequences*, requiring a nested Dim. Although the outer Dim defines a new physical stream, it is discarded because its element size is zero. The stream transporting the nested sequence has $D = 2$ dimensionality bits to encode the three possibilities for every element transported: it is the last element of the inner sequence but not the outer, or it is the last element of *both* sequences, or it is the last element of *neither* sequence.

(d) A type allowing *random* access to an element from a sequence Seq⟨Bits⟨8⟩⟩. We map this to streamspace as: Group⟨Bits⟨L⟩, Dim⟨Rev⟨Group⟨Rev⟨Bits⟨L⟩⟩, Bits⟨8⟩⟩⟩⟩⟩ where L is the number of bits used to represent sequence lengths. The streamlet sourcing the random element *first* provides the length of the sequence on the outermost physical stream, so that the sink knows how large the sequence is (to prevent requesting out of bounds). Then, for every sequence length, the sink may send multiple (hence Dim) requests through a reversed (hence Rev) physical stream. For every request, an element is provided (hence the Group of the Bits and Rev). This describes a streamed RAM interface. The arguments of Group are strictly ordered. To prevent deadlocks, a source *may not assume* that the sink accepts transfers on streams out of the order of appearance as type arguments.

(e) An example of a mapping of the type Variant⟨Prim⟨32⟩, Prim⟨64⟩, Seq⟨Prim⟨8⟩⟩⟩. The first field of the group contains the variant type *tag* to let the sink know what type of instance is contained in the variant. Because the first two potential types are bit

Data type	Tydi container	Definition
Empty	Null	Bits$\langle 0\rangle$ (this is useful increase the tag size for Variant with an Empty type)
Prim$\langle B\rangle$	Bits$\langle B\rangle$	Bits$\langle B\rangle$
Struct$\langle T_1, T_2, ..., T_n\rangle$	ConcatStruct$\langle S_1, S_2, ..., S_n\rangle$	Group$\langle S_1, S_2, ..., S_n\rangle$
	DesyncStruct$\langle S_1, S_2, ..., S_n\rangle$	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$\langle S_1, S_2, ..., S_n\rangle$	Group$\langle$Bits$\langle\lceil \log_2 n\rceil\rangle$, Union$\langle S_1, S_2, ..., S_n\rangle\rangle$
	ConcatVariant$\langle S_1, S_2, ..., S_n\rangle$	Group$\langle$Bits$\langle\lceil \log_2 n\rceil\rangle$, Group$\langle S_1, S_2, ..., S_n\rangle\rangle$
	DesyncVariant$\langle S_1, S_2, ..., S_n\rangle$	Group$\langle$Bits$\langle\lceil \log_2 n\rceil\rangle$, New$\langle S_1\rangle$, New$\langle S_2\rangle$, ..., New$\langle S_n\rangle\rangle$
Tup$\langle n, T\rangle$	ConcatArray$\langle n, S\rangle$	Group$\langle U_1, U_2, ..., U_n\rangle, \forall u \in U, u : S$
	Array$\langle n, S\rangle$	New$\langle S\rangle$
	RATElem$\langle n, S\rangle$	Group$\langle$Rev$\langle$Bits$\langle\lceil log_2 n\rceil\rangle\rangle, S\rangle$
	RATSlice$\langle n, S\rangle$	Group$\langle$Rev$\langle$Group$\langle$Bits$\langle\lceil log_2 n\rceil\rangle$, Bits$\langle\lceil log_2 n\rceil\rangle\rangle\rangle$, New$\langle S\rangle\rangle$
Seq$\langle T\rangle$	List$\langle S\rangle$	Dim$\langle S\rangle$
	Vector$\langle S\rangle$	Group$\langle$Bits$\langle L\rangle$, New$\langle S\rangle\rangle$
	RASElem$\langle S\rangle$	Group$\langle$Bits$\langle L\rangle$, Rev$\langle$Group$\langle$Bits$\langle I\rangle\rangle, S\rangle\rangle$
	RASSlice$\langle S\rangle$	Group$\langle$Bits$\langle L\rangle$, Rev$\langle$Group$\langle$Bits$\langle\lceil log_2 n\rceil\rangle$, Bits$\langle\lceil log_2 n\rceil\rangle\rangle\rangle$, New$\langle S\rangle\rangle$

L is a system-wide constant representing the number of bits to represent indices. RAS stands for random-access-sequence, and RAT for random-access-tuple.

Table 5.3: Overview of Tydi 'container' types.

fields, they can fit into the outermost stream through the Union type, causing the element size to be the maximum of the size of the Bits fields, in this case $E = 64$. Since the third type has a higher dimensionality ($D = 1$), its instances flow over their own physical stream. Whenever the tag exposes that the element is of the third type, the sink must read the rest of the instance from the innermost stream.

(f) A use for New. Instead of mapping the length of a sequence by increasing D, we may choose to map the sequence length as a separate stream. This can be seen as another way of mapping an instance of a Seq into streamspace.

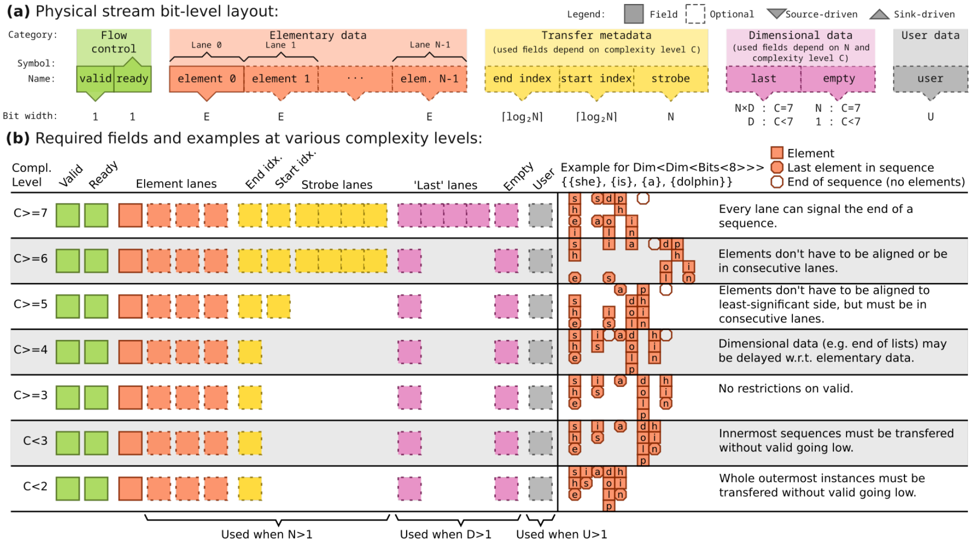
5.3.2. Standard container library

As described in the Background section, software projects provide programmers with pre-defined containers to map data structures to memory. Containers are aliases for combinations of types from the programming language's type system, with some specific access behavior, typically implemented in a standard library. Similarly, Tydi proposes 'containers' for streamspace to represent common data structures. These 'containers' have access behavior associated with them as described by the streamspace type system. Some of these proposed mappings can be found in Table 5.3. The reader is encouraged to draw out some of these similar to the graphs of Figure 5.2, to verify the intuitive hardware-oriented view on data types of the streamspace type system.

5.4. Physical streams

We discussed the streamspace types, and how it determines two properties of physical streams; E, the number of element bits, and D, the number of dimensionality bits to signal the end of a (nested) sequence. We now introduce the bit-level layout of a physical stream and show additional properties of physical streams that are relevant in the context of connecting two streamlet *interfaces* producing and consuming data. When all properties are known, a concrete circuit-level interface can be synthesized.

Physical streams have three additional properties; N, U and C. N is the number of



Every lane can signal the end of a sequence.

Elements don't have to be aligned or be in consecutive lanes.

Elements don't have to be aligned to least-significant side, but must be in consecutive lanes.

Dimensional data (e.g. end of lists) may be delayed w.r.t. elementary data.

No restrictions on valid.

Innermost sequences must be transfered without valid going low.

Whole outermost instances must be transfered without valid going low.

Figure 5.3: Bit-level layout of a physical stream (a) and examples for various complexity levels (b).

elements per transfer. Communicating multiple elements per transfer can be used to scale up the bandwidth of a physical stream at the cost off additional wires. When $N > 1$, the stream has multiple **lanes** over which elements are transported. U is the number of arbitrary **user** bits piggybacking transfers, for whatever purpose. C is the **complexity level** of a stream, that describes the guarantees about the packing of elements into (mainly the temporal dimension of) streamspace. The complexity level can be used to make additional trade-offs about the complexity of the control logic of the interface on both ends of the stream, with minor nuances in area and throughput. Finally, physical streams use the same valid/ready-handshaking mechanism as AXI4-Stream for flow control.

Using these properties, the layout of a physical stream can be seen in Figure 5.3a. The signals fall into the following five categories.

- Flow control; the valid/ready signals for an AXI-like handshake.
- Elementary data; the N elements of size E to be transported in a single transfer, each over their own lane.
- Transfer metadata; used when $N > 1$ to deal with sub-normal transfers (i.e. when not all lanes contain valid data, explained below).
- Dimensional data; `last`, the D-bits to signal the elements are last in some dimension, and `empty`, to signal empty sequences.
- User data; `user`, an arbitrary-size field for custom per-transfer information.

In Figure 5.3b, we also find how the complexity parameter affects the guarantees that may be dropped when increasing the complexity level, effectively changing the number of required signals.

At the lowest complexity level $C = 1$, the source provides the strictest guarantees about the packing of the elements into streamspace. When $N > 1$, a transfer may contain less than N elements (e.g. at the end of a sequence). Requiring elements to be aligned to the least significant lane, the `end index` field signals which lane holds the last valid element. At $C >= 5$, the alignment requirement is relaxed, allowing also a consecutive number of least significant lanes to be invalid, requiring the `start index` as well. At $C >= 6$, any lane may contain valid or invalid elements, introducing the need for a `strobe`. Note that tools using Tydi can automatically insert small combinatorial conversion units in case a sink supports a higher complexity level than a source, to convert the end and start index to strobes. Note that the choice between $C = 5$ and $C = 6$ is rather significant, since when elements are very small but a high throughput is required, strobes require N signals rather than only $2 \cdot \lceil log_2 N \rceil$ signals for the end and start index. Finally, at $C >= 7$, it is furthermore allowed that every element is the last element of a sequence. In other words, a transfer may signal multiple ends of data in some dimensions, and signal multiple empty sequences. Therefore, the `last` and `empty` fields are duplicated for all lanes, linearly increasing the number of wires required for the dimensional data with respect to the number of lanes.

For a detailed discussion, we refer the reader to the Tydi website where the specification is freely available [2].

5.5. Feature comparison

We compare the features of Tydi and existing streaming interface specifications and language abstractions mentioned in the background section. The comparison is shown in Table 5.4. We focus on those features that are novel through this work or common among multiple specifications.

The main difference between Tydi and AXI/Avalon is that Tydi also provides a type system for compound types (e.g. structs and variants) and describes how streams nested within streams must behave, while AXI and Avalon only describe the Tydi equivalent of a single physical stream of primitives or sequences of one dimension. While the Tydi type, and the knowledge that group and union fields adhere strict ordering clearly specifies the interaction, any logical interface with multiple physical streams using AXI or Avalon requires additional specifications.

Transferring higher dimensional information is also undescribed, requiring custom design effort. AXI has a unique feature, called positional bytes that a consumer should not replace in an implied byte-addressable memory being overwritten. This is an implication that is explicitly not used in Tydi, but could simply be supported by wrapping the element in a Group with an additional positional flag bit. AXI and Avalon contain different specific features for element packing, that are both supported through the complexity parameter of physical streams in Tydi. Avalon and AXI contain additional flow control and routing features not described in Tydi, but they can be mapped onto the `user` field.

The comparison between Tydi and the HDL constructs is rather simple, since in all HDLs that we compare, the only thing that is described and abstracted is the valid/ready handshake mechanism. Every other signal of the interface is completely user-defined. While this results in a lot of undescribed features, it provides a starting point for implementations of Tydi in the respective languages.

Table 5.4: Feature comparison of Tydi with existing streaming interface specifications and language constructs

Feature	Specification / language construct			
	Tydi	**AXI** [3]	**Avalon** [4]	**HDLs** [6][7][8]
Intended for	Complex datastr.	Byte packets	Packets, DSP	Handshake only
Elem. size (bits)	$\{1,\infty\}$	8	$\{1,512\}$	$\{1,\infty\}$
Structs	Yes	n.d.	n.d.	Yes
Variants	Yes	n.d.	n.d.	Yes
Stream nesting	Yes	n.d.	n.d.	n.d.
Max. dimensions	∞	1	1	n.d.
Max. data bits per transfer	∞	1024	4096	∞
Container library	Yes	n.d.	n.d.	n.d.
Multiple elem. per transfer	Yes	Yes	Yes	n.d.
Lane control	Aligned, Strobes	Strobes	Aligned	n.d.
Null elements	Yes	Yes	Yes	n.d.
Positional elements	n.d.[†]	Yes	n.d.	n.d.
Back-pressure	Optional	Optional	Optional	Mandatory
Multiplexing	n.d.[‡]	Yes	Yes	n.d.
Credit-based flow control	n.d.[‡]	n.d.	Yes	n.d.
User data; per ...	Transfer	Transfer	Element, Packet	n.d.

Yes: possible, by specification. **No**: not possible, by specification.
n.d.: not described by specification or documentation.

[†] Can be supported by using GROUP with a "don't care" bit field.

[‡] Can be supported with the `user` field.

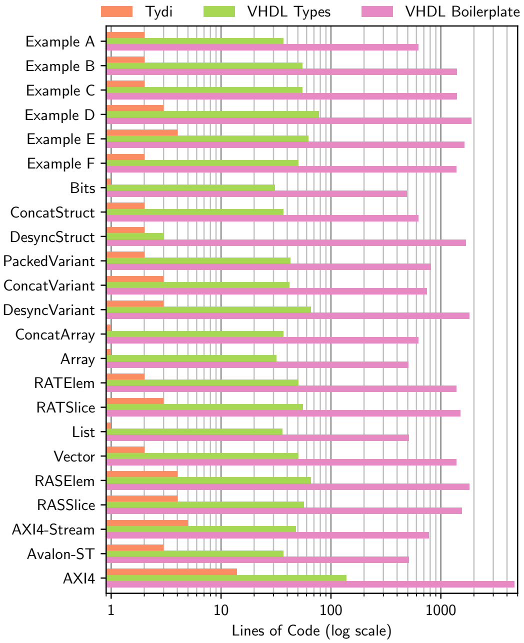


Figure 5.4: Comparison of hardware description effort

5.6. Implementations

We implemented a software utility, found alongside the specification, that serves as a reference implementation. The utility parses files containing declarations of Tydi types as well as streamlets with Tydi interfaces and generates HDL code templates.

Using the templates, users can build libraries of reusable components that have interfaces adhering to the specification. The back-end of the utility is modular, currently generating VHDL, but can be easily extended to other hardware description languages. The generated code consists of a package that contains user-friendly, human-readable VHDL record type hierarchies and readable boilerplate procedures derived from the Tydi types, subjectively not different from how an experienced hardware developer would write them. The generated code can be used to e.g. perform handshakes and decode unions with a single line of VHDL.

To indicate the amount of effort saved by this utility and the Tydi specification and type system, Figure 5.4 compares the size of the input of our utility to its output . A minimum amount of VHDL required are the record type hierarchies, shown in the figure as “VHDL Types” whereas additional boilerplate code is listed as “VHDL Boilerplate”. It is design-dependent how much of this boilerplate code will be used, depending on the procedures and functions used, so this measure gives an upper bound.

We generate code for all types presented in the examples and the container library. Only the Bits generic type parameter is used, only two fields for the containers for Struct and Variant are provided. As Table 5.4 shows, all the other known specifications can be implemented as a Tydi type, which we also did for the whole AXI4 (memory) interface specification. The Tydi equivalent to the HDL constructs is the Tydi Bits type.

Because the code size depends on the physical stream parameters, we generate for $E = 1$, $E > 1$ and all possible values for C, and report the average lines of code for each type. From Figure 5.4, we find that Tydi decreases the required lines of code of all types by an order of magnitude and potentially by another order of magnitude depending on how much of the boilerplate code is used.

We expect to implement additional back-ends for more modern HDLs, such as Chisel and Cλash in the near term. Longer term, the utility can be grown into an HDL of its own to support structural composition of streamlets, followed by behavioral constructs, where the specific rules related to the streamspace type system may be statically or dynamically checked by automated tools. Such a language could borrow from well-studied dataflow languages [9] and from recent implementations of this paradigm [10].

A subset of Tydi is also implemented in the Fletcher FPGA accelerator framework. Fletcher provides a hardware/software interface between data structures in memory and hardware accelerators. Fletcher is built on Apache Arrow, a project that provides a common in-memory data layer for over eleven software languages, preventing the need to serialize/deserialize information between heterogeneous (software) processes, which can incur significant bottlenecks in accelerator systems [11]. Because the data structures that can be expressed in Arrow include nested sequences and variants, existing streaming specifications are not adequate to support all Arrow data types, hence the need for the more advanced streaming specification and infrastructure that Tydi provides. Fletcher translates Arrow types into a subset of Tydi types, and generates the appropriate bus infrastructure and control logic to stream in Arrow data, bridging the gap between hardware and software for any of the languages supported by Arrow.

5.7. Conclusion

While hardware accelerators are becoming increasingly popular, we observed a lack of clear specifications and methods that allow developers to work with complex, dynamically-sized data structures in hardware description languages. We have introduced the Tydi specification, that allows to rapidly express how such structures can be exchanged between components using streaming interfaces, based on an intuitive, hardware-oriented type system. We have shown that by describing components with interfaces based on the type system, the hardware description effort can be reduced by orders of magnitude. Our work enables future integration of the type system into modern existing, or new, hardware description languages, such that the exchange of complex, dynamically-sized data structures between components is as easy to describe for hardware as they are for software today.

References

[1] L. Truong and P. Hanrahan, *A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity,* in *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136, edited by B. S. Lerner, R. Bodík,

and S. Krishnamurthi (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019) pp. 7:1–7:21.

[2] Accelerated Big Data System Group, Delft University of Technology, *Tydi: an open specification for complex data structures over hardware streams,* .

[3] ARM Limited, *AMBA AXI and ACE Protocol Specification AXI3, AXI4, AXI5, ACE and ACE5,* (2018).

[4] Intel Corporation, *Avalon interface specifications,* .

[5] G. Weisz and J. C. Hoe, *CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing,* in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)* (2015) pp. 1–8.

[6] C. Baaij, M. Kooijman, J. Kuper, W. Boeijink, and M. Gerards, *Cλash: Structural descriptions of synchronous hardware using haskell,* in *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools* (IEEE Computer Society, United States, 2010) pp. 714–721, eemcs-eprint-18376.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, *Chisel: Constructing hardware in a scala embedded language,* in *DAC Design Automation Conference 2012* (2012) pp. 1212–1221.

[8] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, *Spatial: A language and compiler for application accelerators,* in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018 (Association for Computing Machinery, New York, NY, USA, 2018) p. 296–311.

[9] W. Thies, M. Karczmarek, and S. Amarasinghe, *Streamit: A language for streaming applications,* in *Compiler Construction*, edited by R. N. Horspool (Springer Berlin Heidelberg, Berlin, Heidelberg, 2002) pp. 179–196.

[10] J. Thomas, P. Hanrahan, and M. Zaharia, *Fleet: A framework for massively parallel streaming on fpgas,* in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20 (Association for Computing Machinery, New York, NY, USA, 2020) p. 639–651.

[11] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee, *Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow,* in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (2019) pp. 270–277.

[12] J. Peltenburg, J. van Straten, M. Brobbel, Z. Al-Ars, and H. P. Hofstee, *Tydi: an open specification for complex data structures over hardware streams,* IEEE Micro (2020).

[13] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, and et al., *The FitOptiVis ECSEL Project: Highly Efficient Distributed Embedded Image/Video Processing in Cyber-Physical*

Systems, in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19 (Association for Computing Machinery, New York, NY, USA, 2019) p. 333–338.

PERSONAL CONTRIBUTIONS, ACKNOWLEDGEMENTS AND FUNDING.
This chapter is based on one article accepted for publication [12]. J. van Straten, M. Brobbel and J.W. Peltenburg made equal contributions to this chapter.

This work is part of the FitOptiVis project [13], funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162.

6
CONCLUSION

In the domain of big data analytics, there is an ever-increasing demand for computational performance. In such increasingly specialized systems, chip resources can be used more efficiently towards achieving a specific computational goal. The recent increase in I/O bandwidth for network, storage, and accelerator interface, is relatively large compared to the increase in CPU performance. This additionally makes the inclusion of heterogeneous accelerators in new systems a more interesting alternative.

Because of the increased complexity of heterogeneous system architectures, special care must be taken to not only allow solutions to have a small run time component, but also a small design time component, since in general, within a given budget, the complete time-to-solution is what matters most.

It is yet to be seen if the time-to-solution for FPGA accelerated systems is economically interesting, especially for widespread use in data centers. In this dissertation, we have explained that because of the many choices a developer has to make at the level of the digital circuit, it is time consuming to produce new solutions that are both functional, exhibit high performance, and integrate well with contemporary software systems — systems that typically enjoy a much higher level of abstraction when working with them.

Until the maturation of productive, open-source tools for FPGA accelerator development, we expect that in the near term, the most successful FPGA accelerated solutions will initially be created by expert developers with hardware design knowledge. To this end, it is helpful to increase the level of abstraction of hardware development tools without the loss of potential performance. This is easier to do in a domain-specific context, since domain-specific architectural knowledge can be implemented in the tools without having to work well for other domains.

On the other hand, accelerators are typically intended to be integrated with software systems. In the domain of big data analytics, the run-time systems used to provide these level of abstraction have various drawbacks that in some cases nullify the advantage of increased I/O bandwidths — one advantage that makes using accelerators increasingly more interesting in the first place.

The drawback of serialization overhead, ultimately caused by collections of data being fragmented in memory so much, that we cannot move data sets over high-bandwidth interfaces without first gathering them in more contiguous regions of memory. The gathering of the data itself cannot be done at throughput similar to the interface, and is therefore detrimental to the efficient integration of FPGA accelerators in big data systems.

The Apache Arrow open-source project was initiated by a community of developers to deal specifically with this challenge, although mainly in the context of software systems. It provides an in-memory format for structured data, especially in the form of tabular data sets, that is column-oriented and highly contiguous. Through the many libraries for various software languages, zero-copy inter-process communication is made possible between heterogeneous software processes.

In this dissertation, we have built on top of the Arrow format to provide a domain-specific hardware development and integration tool-chain, named Fletcher, to enable the same type of functionality for FPGA accelerators. Fletcher generates the complete hardware infrastructure for accelerator designs based on descriptions of the type of Arrow data of interest. The infrastructure provides high-performance, easy-to-use interfaces that match the type of Arrow data. It also contains run-time integration libraries for software, to abstract a large portion of the data and control path from high-level languages down to the hardware kernel performing some accelerated computation. Fletcher is platform-agnostic, allowing developers to target various accelerator platforms without changing any code. The tool-chain therefore integrally improves the time-to-solution when designing and integrating FPGA accelerators with big data systems using the Apache Arrow format.

In experiments, we have shown that the combination of Fletcher and Arrow allows specific applications, such as regular expression matching on a large collection of strings, to improve computational throughput by between 1.3× and 49×, specifically by preventing serialization from taking place from less optimal in-memory formats that are the default in some language run-time systems and libraries. The design effort is significantly reduced through the use of Fletcher, although how much is hard to quantify properly due to the large human dimension in such a measurement. It is perhaps indicative that it only takes tens of lines of code to express the Arrow data types, in turn generating all code related to the high-performance infrastructure feeding the accelerated kernel implementation, which can be in the order of thousands of lines of code.

Using the Fletcher framework, we have furthermore explored several applications in the domain of big data analytics that have a high FPGA acceleration potential, such as Hidden Markov Models in a genomics context, and decoding the widely used Parquet storage format, where improved performance over a traditional system with a general purpose processor was observed in both cases.

Driven by the desire to reduce the design time of FPGA accelerators, we have applied lessons learned in a final contribution where we extend transport Arrow data structures over streams to a more generic approach in the form of a streaming interface specification and type system named Tydi. In software, standard libraries of well-developed languages provide all sorts of containers for data, with specific access behavior, that are highly reused when developing applications and sharing designs. We may think about a string object, for example. While it is in the typical case absolutely clear what that means in software languages, and especially how one passes a string from one function to another, in hardware description languages, there is no such thing as a default string, causing developers to spend a lot of time making choices about how to represent and transport it across a design. Tydi allows such a software-inspired view on data structures, but in an explicitly hardware-oriented manner. Tydi is a more formal extension of the more ad hoc Fletcher streaming interface protocol, and does allow a clear and reusable definition of interfaces that exchange dynamically sized, nested data structures, such as e.g. a list of strings. We have implemented some tools that generate boilerplate code surrounding the types, that indicate orders of magnitude reduction in design effort when working with complex, dynamically sized data structures in hardware.

Research questions

Over the course of this dissertation we have developed answers to several questions posed at the end of Chapter 1. We reiterate the questions, summarize the answers and give an outlook on some potential future research in the direction of the question.

What challenges arise from the desired merger of big data systems software and FPGA accelerators? On the side of FPGA accelerators, we observed a lack of portability. Furthermore, a large portion of the design entails infrastructure and interfacing. On the side of big data software systems, we observed complex run-time systems, hardware-unfriendly data lay outs, ultimately causing significant (de)serialization overhead. We have described how the approach of the Apache Arrow open-source project, to place the data in memory in a column-oriented fashion, as contiguous as possible, may help to overcome the challenges on the big data analytics side, and how Fletcher makes use of the merits of Arrow to automate a large portion of the infrastructure and interface design, furthermore providing portability.

In the future, as the increase in performance of traditional processing systems continues to slow down, more of the existing software systems that were not originally designed for big data analytics will be slowly replaced. It would be useful if the inclusion of heterogeneous components, such as GPGPUs and FPGA accelerators, is taken into consideration from the drawing board of these new systems, both from the hardware perspective and the software run-time systems and programming languages. Lessons may be learned from initial efforts to specify such systems, as was done in e.g. the Heterogeneous Systems Architecture specifications, but more effort is required to raise the level of abstraction of using such components to a level a majority of application developers in big data analytics would be comfortable with.

What of an FPGA accelerator design and software interface can be automated in the context of big data systems? Based on Apache Arrow, we have contributed the Fletcher FPGA acceleration framework. The framework automates a large portion of the infrastructure and interface design, where easy-to-use, high-performance interfaces are presented to users of Fletcher at both the level of the hardware and at the level of the software where the accelerator is to be integrated.

In the future, several improvements could be made to Fletcher, including:

- **Improved resource utilization:** Especially when accelerator kernels need to work on many columns, a large number of infrastructural resources are generated, while it is likely not all of them are fully utilized when the system is operational. Based on measurements of running applications with the profiling feature, it would be interesting to explore methods to automatically optimize the design by allowing specific resources to be regenerated with different dimensions or by sharing them across multiple interfaces based on profiling outcomes. In the best case, these type of data-centric architectural improvements could be implemented while the system runs, fine-tuning the performance without the need for a developer to interfere.

- **Extend the number of supported languages:** More of an engineering exercise, it would be useful to support more of the languages that Arrow supports, such that high-performance integration between FPGA accelerators and more software languages is made possible.
- **Extensively quantify the design effort saved:** We have so far only reported indications for the decrease in design effort through measurements of lines-of-code. While the measurement is very precise, design effort also knows a large human dimension that requires a different type of experiment and a large set of test subjects to be more properly be quantified.
- **Closed-loop architectural optimization:** Making use of the stream profiling capability, it would be interesting to investigate a form of closed-loop stream parameter optimization to increase throughput, based on statistical profiles about the data streams entering the system. Methods should be developed to introduce profile-based parameters that influence how computational kernels and the data streams they interface with may be optimized.

(How) can a platform-agnostic environment be created in the currently highly vendor-specific context of FPGA accelerator design? We have described the architecture of Fletcher, where for all platforms, there is one common interface that must be adhered to on both the software and hardware side. After creating various platform-specific hardware designs and software libraries to convert platform-specific interfaces to the common interface, it is easy to port designs between various systems. All applications that were measured on various FPGA accelerated platforms have made use of this functionality. No code application-specific code has to be changed in order to port a design to another platform.

In the future, especially in the case of FPGA accelerators, where high-bandwidth memories have recently become available, special care must be taken to allow novel technologies to be effectively used. The potential of such advancements should not be nullified by conflicting properties of the common interface.

What applications can benefit from the features of the Fletcher framework? Fletcher provides streaming interfaces that are most effective when sequentially accessing columns of tabular data sets in the Arrow format. Preventing serialization overhead is a major advantage of the Arrow framework, and since Fletcher is built on top of that, any applications that suffer from serialization overhead and that are acceleratable by FPGA can benefit from the Fletcher framework.

In the future, it is interesting to investigate the possibility to specialize the framework further towards more specific application domains, e.g. database queries or machine learning.

Can we decrease the complexity of describing interfaces between hardware components that exchange complex data structures? We have described the need for a more agile hardware development experience, specifically in the context of decreasing the design time for FPGA accelerators in big data systems. Data structures in that context are often dynamically sized (e.g. strings) and can be complex (e.g. nested lists), but no clear specifications are publicly available that describe how such data structures should be transported, hence we have asked this question. We have developed the Tydi streaming interface specification, type system and an initial set of tools to quickly generate HDL templates to use such data structures. We have indicated the design effort saved through lines of code, although such a measurement is subject to the disadvantages previously described in this section, and should be more properly quantified in the future, when the tools surrounding the specification are more mature and when more applications have been developed.

In the future, it would be worthwhile to expose the benefits of Tydi in modern HDLs, such as Chisel, Spatial, or Cλash, where libraries or other constructs may be introduced that allow interfaces to be specified according to the Tydi type system. It would also be interesting to investigate creating a new dataflow language to describe hardware using Tydi. Initially, this would be easiest to develop in a form where only structural hardware may be described. After studying the implications on the access behavior of data structures through Tydi streams, it may also be feasible to introduce behavioral constructs, that can then be statically checked for correctness, discovering potential errors as early as possible, reducing the development time of digital circuits processing complex data structures.

How can FPGA accelerators be efficiently integrated with contemporary big data systems software? Finally, we summarize an answer to the main question of this dissertation. Since we have defined ’efficiently’ to mean ’with the lowest time-to-solution within a given budget’, balance should be sought between a decent time to design FPGA accelerated systems and the run time of FPGA accelerated systems. Fletcher currently helps by automating large portions of the infrastructure and interface design for tabular data structures, reducing the design effort, and by the grace of Apache Arrow, preventing serialization overhead, increasing the interface throughput. As such, Fletcher integrally helps to improve the efficiency of integrating FPGA accelerators with big data systems software.

On a final note, since FPGAs have the inherent technological disadvantage of lower clock frequencies and circuit overhead, it is likely that only the most specialized and most highly optimized solutions form a competitive enough alternative to other platforms such as CPUs and GPGPUs. We therefore argue for HDL design flows that do not favor abstraction at the cost of performance like in the current state of HLS flows, and furthermore argue that HDLs themselves should be improved to make the design time lower using meaningful hardware-oriented abstractions, such as Tydi. Because creating new languages takes considerable effort, solutions such as Fletcher help on the near-term to deal with the problem in a domain-specific manner, but there is a large window of opportunity to decrease HDL flow effort in general. From a community perspective, it would also be useful if in general FPGA development tools become more friendly towards open source, such that it becomes easier to exchange and reuse designs, and to provide portable solutions that work across FPGA-enabled cloud systems everywhere.

List of Publications

Journal papers

2. J. Peltenburg, J. van Straten, M. Brobbel, Z. Al-Ars, and H.P. Hofstee, *Tydi: an open specification for complex data structures over hardware streams* Accepted to appear in the IEEE Micro Special Issue on Agile and Open-Source Hardware (July/August 2020).

1. J. Peltenburg, J. van Straten, M. Brobbel, H.P. Hofstee, and Z. Al-Ars, *Generating high-performance FPGA accelerator designs for big data analytics with Fletcher and Apache Arrow*, Under minor revision for Springer Journal on Signal Processing Systems (2020).

Conference papers

6. J. Peltenburg, L. van Leeuwen, J. Hoozemans, J. Fang, Z. Al-Ars, and H.P. Hofstee, *Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA*, Preprint.

5. J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and H.P. Hofstee, *Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow*, in 29th International Conference on Field Programmable Logic and Applications (FPL) (2019) pp. 270–277.

4. L. van Dam, J. Peltenburg, Z. Al-Ars, and H.P. Hofstee, *An Accelerator for Posit Arithmetic Targeting Posit Level 1 BLAS Routines and Pair-HMM*, in Proceedings of the Conference for Next Generation Arithmetic 2019, CoNGA'19 (Association for Computing Machinery), New York, NY, USA, 2019.

3. J. Peltenburg, J. van Straten, M. Brobbel, H.P. Hofstee, and Z. Al-Ars, *Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow*, in Applied Reconfigurable Computing, edited by C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz (Springer International Publishing, Cham, 2019) pp. 32–47

2. J. Peltenburg, A. Hesam, and Z. Al-Ars, *Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?* in High Performance Computing, edited by J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf (Springer International Publishing, Cham, 2017) pp. 220–236.

1. J. Peltenburg, S. Ren, and Z. Al-Ars, *Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm*, in 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM) (2016) pp. 758–762.

TALKS

3. J. Peltenburg, *The Fletcher Framework: Bringing Apache Arrow to FPGA*, OpenPOWER North America 2019, San Diego, USA

2. J. Peltenburg, *The Fletcher Framework for Programming FPGAs*, OpenPOWER Summit 2018 – Europe, Amsterdam, Netherlands

1. J. Peltenburg, *Integrating Apache Arrow and FPGAs on OpenPOWER*, OpenPOWER Summit US 2018, Las Vegas, USA

OPEN-SOURCE PROJECTS

3. Tydi: an open specification for complex data structures over hardware streams, `https://github.com/abs-tudelft/tydi`

2. Cerata: a Hardware Construction Library written in C++17, `https://github.com/abs-tudelft/cerata`

1. Fletcher: A framework to integrate FPGA accelerators with Apache Arrow, `https://github.com/abs-tudelft/fletcher`

OTHER PAPERS

2. B. Zhu, N. Ahmed, J. Peltenburg, K. Bertels, and Z. Al-Ars, *Diminished-1 Fermat Number Transform for Integer Convolutional Neural Networks*, in IEEE 4th International Conference on Big Data Analytics (ICBDA) (2019) pp. 47–52.

1. T. Ahmad, N. Ahmed, J. Peltenburg, Z. Al-Ars, *ArrowSAM: In-Memory Genomics Data Processing Using Apache Arrow*, in IEEE International Conference on Computer Applications & Information Security (ICCAIS) (2020)

Acknowledgments

Without the help and support of many, this work would have never been completed, and I owe my sincere gratitude to each and every one of you.

Zaid, your unending and contagious enthusiasm have brought me to where I am today. Thank you for the opportunity to make this journey, where you have shown to be very open-minded, always willing to listen, to give advice, motivate people, and stand up for them. We learned a lot and shared a lot of great moments on this journey together, and I will never forget them.

Peter, I tremendously enjoy working with you and learning from you. You are one of the smartest and most wholesome people I know. The other day I watched some movie where they had to select the best of humanity to save the planet - I would select you. Thank you for taking great care of me on this journey. It was (and still is) an amazing privilege.

Jeroen, it is thanks to you that anything we've built during the course of the thesis actually works. Without your contributions and ideas, none of our projects would be in the state they are today. I'm sure the time required for AI to get up to par with humans has significantly increased by you. Thanks buddy!

Matthijs, your unrelenting willingness to help has made working with you an incredibly pleasant experience. It was very educational for me to learn all the zoomer technologies from you. Thanks a million. I hope we may work together for many years to come (and I swear, not just because I want to taste more of your excellent beers).

Joost, I'm very glad you came back to our group. This helped me to regain a positive attitude near the end, something that is contagious about working with you. I'm happy we got to do a few papers together before the end!

Baozhou, my bright-minded friend. As you know I was a 'crazy teacher' for a while, but the rate at which you improve your knowledge and skills, I have never seen before. I hope we can one day find a beer for you that does not taste like water.

Dorus and Robin, we haven't worked together much on technology, but we sure drank a hell of a lot of coffee and beers together. I'm sure this is going to cost me some years at the end of my life (this something you think about a lot when you get old, like me). But let me say it has improved the quality of the recent years far beyond the quality of those that I will lose at the end, so thank you!

I would like to express my sincere gratitude to the MSc students I have intensely worked with, and who have graduated alongside this thesis, and have provided valuable contributions. In chronological order: Tudor, we made a nice trip, and your project was at the forefront of many things we are doing in the group right now. Ahmad, I wish you all the best in finishing your own PhD, you are a very smart guy, and I'm glad you were able to find such a cool place to work in. Laurens, thank you for your contributions, being one of the first users of our project. Please stop killing butterflies. Lars W., thank you for your valuable contributions. You are a great engineer who never failed to surprise us in a positive way. Lars v.L., you are what I usually classify in my mind as an all-round excellent guy, you can do anything. You will go far in the industry! Erwin, you are a very smart guy that made me smile a lot. I'd like to thank all other MSc students that I have worked with as well, some of who are making valuable contributions as I'm writing this, Konstantinos, Osman, Akos, Mirza, Fabian, and many others.

I would like to thank my other PhD colleagues that I have somehow interacted with. Shanshan, it was a real pleasure to work and travel to China with you, and learn a lot about the culture. Jian, I enjoyed working with you and learning from your insights on databases. I wish you all the best with your great new position, we'll keep in touch. Erik, Nauman, Tanveer, Leon, Haji, Mengfei, Kati, and others, thanks for working with me, and chatting about generic stuff on regular occasions. Daniel, Saša, Satoshi, Michel, Augusto, and the others, I will remember that trip to China forever.

My sincere thanks to all of Q&CE's staff, Koen, Stephan, Said, and especially Erik who was always willing to help create practical and robust solutions, and who provided us with the latest and greatest. Thanks to Lidwina and Joyce for taking great care of us, and to Trisha and Laura for helping with the final steps. Thanks to the Graduate School for the interesting DE program and thanks to Wolter for giving advice.

I thank IBM for having me over in Austin for a couple months, especially Jinho, Bedri, Eric and Chris. Thanks to my hosts and friends in Austin, Frieda, April and Jake, Ruby, and Frank and Lisa, for making my time there unforgettable.

Thanks to Xilinx, especially Cathal and Patrick, for being so supportive of our work and pointing us in very interesting directions. Thanks to the OpenPOWER foundation, for letting us give lots of talks and meet a lot of interesting people.

My gratitude to the members of the committee for taking the time to read the thesis, and making the time for the defense.

People I met along the way of my professional life, that taught me valuable things I used during the PhD, deserve my gratitude. Thanks Emile and Clemens for teaching me many (soft) skills and to always stay positive. Thanks to all the other colleagues from my time at the HR and TBP.

Lots of friends from my personal life deserve my gratitude, pulling me ♫ back to life, back to reality ♫, once in a while, thanks guys. I'm not sure if the department will pay for the extra pages it would take to thank you, but you know who you are. Thanks Piet and Ludwig for wanting to be my paranymphs, too bad the 'rona got in the way, but happy to see you in the audience (as it stands at the time of writing).

Thanks mom and dad for creating me and raising me with love <3, in the end a very important contribution to this thesis as well. Also thanks to the rest of the family (in law) for being supportive and enduring my bad jokes all the time.

Emma, a new, far greater and incomparably beautiful challenge awaits us now, one that we will journey on together. I don't think there is a language in the world that could describe the joyful feeling I owe to you and that little 'challenge' sleeping on your lap as I type this.

Curriculum Vitæ

Johannus Willem Peltenburg

March 9, 1986	Born in Sommelsdijk, The Netherlands

Education

2009–2014	*MSc Computer Engineering* Delft University of Technology, Delft, The Netherlands
2005–2009	*B. Eng Electrical Engineering* Rotterdam University of Applied Science, Rotterdam, The Netherlands
1999–2004	*HAVO (N&T)* Regionale scholengemeenschap Goeree-Overflakkee, Middelharnis, The Netherlands
1990–1998	*Elementary school* Bosseschool, Middelharnis, The Netherlands

Work

2009–2016	*Teacher* Rotterdam University of Applied Science, Rotterdam, The Netherlands
2007–2009	*Hardware/Electronics Design Engineer* TBP Electronics, Dirksland, The Netherlands