Solving convex optimization problems on FPGA using OpenCL

Martijn Berkers

Q&CE-CE-MS-2020-02

Abstract

The application of accelerators in HPC applications has seen enormous growth in the last decade. In the field of HPC demands on throughput are steadily growing. Not all of the algorithms used have a clear HW architecture which performs the best. Our work explores the performance of different HW architectures in solving a convex optimization problem. These algorithms are a sequence of dependent operations making it an interesting use-case because parallelism is not easily found. Our work focuses on a use-case of an on machine computational model present in ASML, we explore the acceleration of a quadratic programming Active-Set algorithm on dedicated hardware. There are libraries available to do this on both the CPU and GPU, while nothing is available for the FPGA. Our work focuses on filling this gap by implementing the algorithm using a high-level abstraction parallel programming language in order to ease development for FPGA accelerators. We use the Intel FPGA SDK for OpenCL framework to evaluate the performance trade-offs involved with FPGA acceleration and compare the performance to both the CPU and GPU using library functions. To fit FPGA architecture the algorithm is converted to a dataflow algorithm to enable streaming of data between kernels. The implementation leverages the features introduced in the Intel FPGA SDK for OpenCL framework to stream data using on-chip low-latency communication between kernels. We demonstrate that such a complicated algorithm can efficiently be implemented using the OpenCL framework. Our implementation achieves competitive performance compared to optimized library function on both the CPU and GPU. The OpenCL framework allows for easy design space exploration. We have explored different optimization strategies. The execution time of the final FPGA implementation is 3.5x and 1.2x longer than the CPU and GPU respectively in double precision floating-point. If the accuracy of the FPGA implementation is reduced to single precision there is a speedup of 2.2x in execution time compared to the double precision variant. Higher throughput can be achieved by duplicating the implementation. With the current size of the algorithm, two additional copies are possible. A handcrafted implementation could further improve the FPGA performance by manually managing local memory structures and reusing processing elements. However, significantly fewer lines of code are required, and a significant reduction in development time is achieved by using the OpenCL framework compared to traditional hardware description languages.

Solving convex optimization problems on FPGA using OpenCL

by

M. D. Berkers

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday February 27, 2020 at 1:00 PM.

Student number:	4223438	
Project duration:	May 10, 2019 – February 27, 2020	
Thesis committee:	Dr. ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. ir T. G. R. M. van Leuken,	TU Delft
	Ir. S. C. van der Vlugt,	ASML, supervisor

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

TU Delft Delft University of Technology

Preface

This thesis is the final product of my work at ASML over the last 9 months, titled *"Solving convex optimization problems on FPGA using OpenCL"*.

I would like to thank many people that were involved in or contributed to this work. First, Steven van der Vlugt for his supervision. He always provided valuable insights and a listening ear whenever I needed it. His help shaping the story has helped immensely. Second, Zaid Al-Ars for always questioning the work to see that no stones were left unturned. Third, John Wagensveld for his guidance and remarks to provide a new perspective. Last but not least, Lennart Noordsij for his continued support as a friend and colleague. His help by reviewing documents and being there to bounce ideas around has been invaluable.

I would like to thank my friends and family. This work would not have been possible without their support. They provided me with the opportunity to take my mind of work whenever I needed it.

I would also like to thank ASML for providing me with this opportunity and a great place to work. My time at ASML has not only been educational but also fun. Mostly because of the great colleagues who were always there to enjoy a coffee with. Especially Vivek Jaiswal, his company and jokes helped me feel at home.

M. D. Berkers
Delft, February 2020

Contents

List of Figures

List of Tables

Abbreviations

ALM Adaptive Logic Module

API Application Program Interface

ASIC Application Specific Integrated Circuit

AVX Advanced Vector eXtensions

BLAS Basic Linear Algebra Subprograms

BRAM Block RAM

BSP Board Support Package

CLB Configurable Logic Block

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CU Compute Unit

CXL Compute eXpress Link

DDR Double Data Rate

DMA Direct Memory Access

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processor

DUV Deep UltraViolet

EUV Extreme UltraViolet

FA Full-Adder

FF Flip-Flop

FIFO First In First Out

FLOPS/s FLoating-point OPerationS per second

FPGA Field-Programmable Gate Array

GCC GNU Compiler Collection

GEMM GEneral Matrix-Matrix product

GEMV GEneral Matrix-Vector product

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HDL Hardware Description Language

HLL High-Level Language

HLS High-Level Synthesis

HPC High Performance Computing

IC Integrated Circuit

II Initiation Interval

ILP Instruction Level Parallelism

IoT Internet of Things

IP Intellectual Property

LAPACK Linear Algebra PACKage

LUT Look-Up Table

MAC Multiply-ACcumulate

MIMD Multiple Instruction Multiple Data

MKL Math Kernel Library

OpenCL Open Computing Language

PE Processing Element

PoC Proof of Concept

PR Partial Reconfiguration

qpAS quadratic programming Active-Set

RAM Random-Access Memory

RTL Register-Transfer Level

SIMD Single Instruction Multiple Data

SoC System on Chip

SPD Symmetric Positive Definite

SPMD Single Program Multiple Data

VHDL Very High Speed Integrated Circuit Hardware Description Language

1 Introduction

With Moore's law [38] coming to an end, there is a pressing need to find an alternative to simply adding more transistors to the traditional **Central Processing Unit** (CPU), especially in **High Performance Computing** (HPC) applications. The increased transistor scaling is not the only challenge. The power wall, a consequence of Dennard scaling [9] is another limitation in traditional CPU architectures shifting the focus from single core performance to multi-core design. Dedicated hardware architectures such as the **Graphics Processing Unit** (GPU) and **Field-Programmable Gate Array** (FPGA) in collaboration with the existing CPU infrastructure are being adopted into HPC clusters. Largely because of the accumulation of data and increased complexity of new applications in the field of HPC. The GPU is interesting because of the massive number of cores and high memory bandwidth, and the FPGA because of the power efficiency and runtime reconfigurability. Cloud computing services such as Amazon Web Services [1], Microsoft Azure [37] and Nimbix [41] are already offering customers FPGA and GPU instances to accelerate their applications that need high connectivity or throughput.

A common trend in the field of HPC is that the data size and complexity of the algorithm are growing faster than the computation power provided by traditional architectures. One solution is to increase the number of CPUs creating a cluster of nodes. This introduces additional communication overhead to handle the division of work. Moreover, cooling the heat produced by these machines is starting to become a problem, requiring more efficient solutions than just adding more nodes. A single accelerator might achieve the same performance as multiple CPU nodes. Depending on the application a GPU can exploit the data parallelism and **Instruction Level Parallelism** (ILP) of applications. This takes advantage of the high number of cores and memory bandwidth available on the GPU. For the FPGA properties which can be used to get performance are harder to define. Having high ILP that could allow the FPGA to create deep pipelines to achieve a high throughput. Another option is adding more **Compute Unit**s (CUs) to exploit data parallelism, this uses additional resources to achieve higher throughput.

If the decision is made to use an accelerator, the application still has to be ported to that specific hardware architecture. Using the GPU as an accelerator is not trivial, to properly take advantage of the architectural properties, experience and knowledge is needed. The interfaces that can be used on the GPU are **Open Computing Language** (OpenCL) and CUDA (NVIDIA only) [43]. Optimized custom GPU implementations require training and experience to develop. As the number of people with this knowledge is limited, alternative methods which include optimized libraries are required. This is even more relevant for an FPGA accelerator, where the standard programming languages are **Hardware Description Language**s (HDLs), mainly **Very High Speed Integrated Circuit Hardware Description Language** (VHDL) and Verilog. A completely different programming model is used by HDLs compared to more traditional languages such as C and FORTRAN. To bridge the gap between hardware and software, **High-Level Synthesis** (HLS) and **High-Level Language** (HLL) tools translate a high-level language to an HDL are increasingly supported by the largest FPGA manufacturers. Programming languages such as C and C++ can now be used to program FPGAs [58]. Additionally, acceleration frameworks such as OpenCL are now supported by both Xilinx [57] and Intel [19].

1.1. Context

ASML is world leader in high-end photolithography machines. They are used by large **Integrated Circuit** (IC) manufacturers to keep up with the increasing demand of ICs in domains such as smartphones, servers, storage, **Internet of Things** (IoT) and automotive [33]. ASML provides chip manufacturers such as TSMC, Samsung, Toshiba and Intel with machines that have to provide a steady production for 15-25 years. With the new generations of the **Deep UltraViolet** (DUV) and **Extreme UltraViolet** (EUV) machines the overlay and critical dimension requirements keep increasing to allow for even smaller and more efficient ICs [36].

Metrology computational models adjust the machine for exposing the wafers, taking into account the imperfections and deviations between each exposure. To allow for even smaller transistors these models foresee a growth in data and complexity. This increases the computational requirements while at the same time time-budgets are tightened to increase wafer throughput. Alternative solutions to the classical CPU could prove important in dealing with future computational challenges. In the field of HPC there are three main options when choosing **Commercial Off-The-Shelf** (COTS) hardware: CPU, GPU and FPGA. The CPU is usually favored, because of ease of programmability and portability, but lacks in terms of raw **FLoating-point OPerationS per second** (FLOPS/s) and memory bandwidth for massively parallel applications. Because of this there is increased interest in the GPU and FPGA to offload applications from the CPU to a better suited architecture. One of the requirements within as using accelerators becomes increasingly popular is that software developers have be able to take advantage of this acceleration. Because of this, using high-level languages such as C, C++, OpenCL and CUDA and HLS tools provide a middle ground between hardware and software. Knowledge about the architecture is still needed, but development is faster and porting an application to a new platform is easier. This study was proposed to research the use of an FPGA accelerator for HPC applications. For the GPU one such study was already conducted within ASML by Bamakhrama et al. [2], where they show that GPU acceleration could provides a solution to the increasing computational requirements of a model used in EUV lithography.

This thesis aims to broaden the knowledge within ASML on using the FPGA as an accelerator and provide insights in the trade-offs inferred by the chosen framework.

1.2. Industrial use case: Convex optimization algorithm

Many computational models used at ASML use linear solvers to find the correct solution from a large solution space. Our use-case focuses on a Convex optimization problem, the optimal solution is found using the **quadratic programming Active-Set** (qpAS) algorithm. It computes an optimal step in the feasible region, this means that the algorithm can be stopped at any point. The solution at that time will be feasible within the boundaries, even if all the requirements are not satisfied. The algorithm keeps taking steps until the solution that satisfies all conditions is found. There is no indication of how many iterations are required before the optimal solution is found. The mathematical operations in the iterative region include matrix-vector multiplication, matrix multiplication, Cholesky factorization and forward/backward substitution. The input data size for these operations changes between iterations because the Active-Set method is used, where constraints are added or removed each iteration.

An FPGA is considered a suitable candidate to accelerate the algorithm because it offers more parallelism than a CPU and at the same time handles sequential execution more efficiently than a GPU. Generally, FPGAs are better suited for regular execution by creating a compute pipeline. There are multiple entry-levels for implementing an application on the FPGA. From low-level, working on connecting gates and registers to high-level, where C-code can be used.

Typically software engineers implement these computational models in high-level languages that run on the machine. The scope of this thesis is to get a better picture of what tools such as HLS and OpenCL for FPGA are capable of. For this study the OpenCL SDK for FPGA [19] framework was chosen as the entry-point. Intel advertises that this framework allows software developers to accelerate applications on FPGA.

The main research goal of this thesis is to investigate the feasibility of using FPGA accelerators with OpenCL as the abstraction level, as well as the performance trade-offs on the FPGAthe qpAS model using this high-level abstraction. To achieve this goal we define several research questions.

- What parts of the algorithm are candidates for acceleration?

- Can the algorithm be implemented on the FPGA using a high abstraction approach?
- What are the performance trade-offs on the FPGA using the OpenCL framework by Intel?
- What is the performance of the FPGA compared to a CPU and GPU using COTS libraries?

To answer these questions we will first profile and analyze the algorithm. Because the CPU is the default deployment hardware, that is what we use for profiling and reference. After the bottlenecks and hot spots are identified, we will propose several hardware and platform based solutions which aim to solve bottlenecks and execute hot spots in parallel. The scope of this thesis is using a high-level abstraction to accelerate linear algebra algorithms using the FPGA. To draw a fair conclusion about the benefits of an FPGA with this high-level approach, it is compared to a CPU and GPU implementation. A basic comparison between the CPU, GPU and FPGA in terms of performance executing hot spots and the capabilities of dealing with bottlenecks will provide the input for this conclusion. The high-level abstraction chosen for the FPGA is the OpenCL SDK for FPGA framework by Intel [19]. For the GPU and CPU, CUDA and MKL with COTS libraries are used. The goal is not to get the an optimized implementation on the FPGA, but use an approach that software developers can use. With the results the performance and deployment trade-offs between the FPGA and GPU when offloading the qpAS algorithm using a high-level abstraction are evaluated.

1.3. Synopsis

The thesis is structured as follows: in Chapter 2 the necessary background information is provided about hardware architectures, OpenCL, OpenCL SDK for FPGA and the algorithm. In Chapter 3 the related work is evaluated about using accelerators with high-level languages for linear algebra HPC applications. There we will discuss how our research differs from existing work. In Chapter 4 we analyze the algorithm and propose solutions. In Chapter 5 the solution according to the scope is implemented. In Chapter 6 the results of the implemented solution are evaluated and compared against the current implementation. In Chapter 7 the thesis results are concluded and recommendations are made for further work on the algorithm and deployment strategies.

2

Background

In this chapter all the relevant background information is given. In Section 2.1 to 2.4 the compute platforms are explained. In Section 2.5 the programming framework OpenCL is explained and its use to target different hardware architectures are highlighted and compared. The FPGA manufacturers each have their own platforms using their own tooling. In Section 2.6 the OpenCL SDK for FPGA is highlighted. Finally, in Section 2.7 information about the routines that are used for the project is given.

2.1. Compute platforms

There are three main compute platforms available: CPU, GPU, FPGA. Where the FPGA is least used in terms of HPC applications. With the recent acquisition of Altera by Intel and continued research by Xilinx, there is renewed interest in using the FPGA as an accelerator. As seen in Figure 2.1, the CPU provides flexibility more than the other two. Traditionally most of the computing was done on one or multiple CPUs. The FPGA has also been around for a long time, but as this usually involved programming on a low abstraction using HDLs, it would take a long time to design each function on an FPGA. This was one of the main reasons it was not commonly used in HPC applications, where flexibility and performance are the requirements. In recent times GPUs emerged as an alternative. While they were not created for scientific computations, this naturally flowed from the design, which has a high number of cores optimized for integer or floating-point arithmetic.

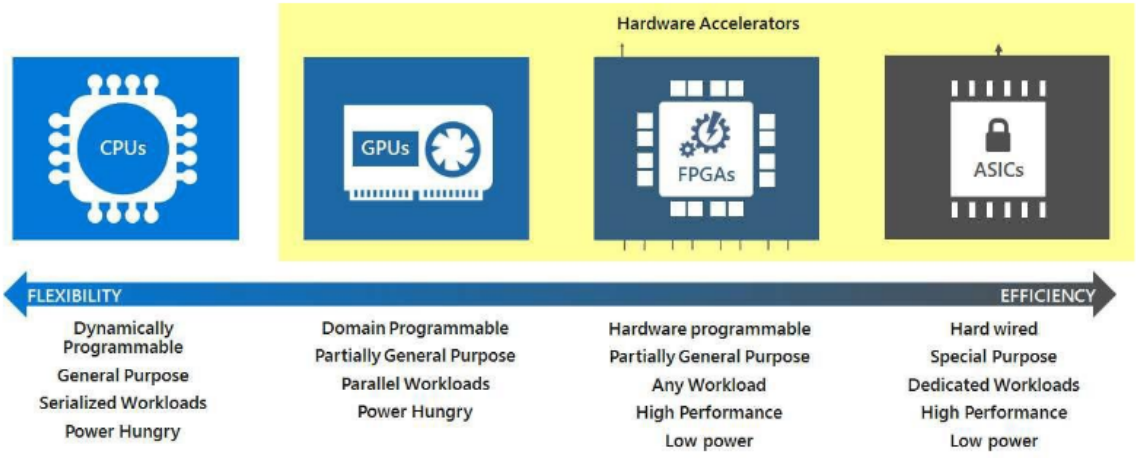

Figure 2.1: Compute platforms in terms of flexibility and efficiency [7]

2.2. Accelerator offloading

If either the FPGA or GPU is used to compute a specific portion or whole algorithm we can consider this device an accelerator. Applications and routines can be offloaded to this device to reduce the load on the host or improve performance in execution time or throughput. There are several considerations to make while offloading tasks to an accelerator. Data is present on a host device and has to be moved to the accelerator when it is needed for computation. The program also has to be transferred or programmed onto the device, depending on whether it is hardware configured (FPGA) or programmable (GPU). These steps take time, and can be considered overhead compared to a CPU only implementation. In [55] the authors use **Partial Reconfiguration** (PR) compute kernels on FPGA to explore offloading computationally hard routines to an accelerator. They use a Microblaze [59] core in combination with a FPU or **Basic Linear Algebra Subprograms** (BLAS) tile. They determine a point where the improvement in computation time becomes more than the overhead incurred by transferring the data. They conclude that routines that contain more than 100,000 operations are worth moving to an accelerator. This means that small input data structures will not be worth offloading to an accelerator. Not only computational applications can benefit from an accelerator. In [27] the authors use FPGAs for cloud-computing applications like Memcached and achieve a speedup of up to 32 compared with CPU. The benefit for cloud-computing applications is energy consumption, an FPGA is more energy efficient. In data centers where these applications normally run, reducing power while keeping performance is commercially interesting.

The interconnect between the host and device plays a role when offloading routines or applications. How important that role is depends on the application, when the data size that is send over the link is small it will not play an important role in performance. The authors of [11] evaluate the use of FPGAs to accelerate in-memory database systems. One of the bottlenecks identified in this paper is the bandwidth between the host system and accelerator. There are two ways of solving this problem, reducing the data or increasing the throughput. NVIDIA already has NVLINK that provides a higher throughput between their GPUs and compatible devices. Another option is updating the PCIe link to a newer version, where throughput doubles each generation. There are layers possible on top of PCIe such as the **Compute eXpress Link** (CXL) that provides memory coherency between host and device memory.

One benefit of using an FPGA as accelerator compared to GPU is the flexibility in inter-FPGA and network capabilities. It is possible to directly transfer data via Ethernet or InfiniBand into the FPGA fabric [29]. Using these links provides low-latency high bandwidth data-sharing between the FPGA and other network attached devices. In [12] the authors show that scalability of a multi-FPGA system using Ethernet is a viable option. On a 40 Gbit/s link they achieve a throughput of 29.77 Gbit/s with a latency of 950 ns.

2.3. GPU

A GPU is a type of integrated circuit originally made for image processing in computer and consoles. Because the number of pixels in a screen is large it usually features a higher number of cores than a CPU but these cores also have lower performance and are less capable. GPUs are designed for vector instructions and thus well suited for linear algebra workloads.

GPUs come in two varieties, integrated GPUs that are contained in a dedicated part on a CPU with lower core count but shared memory, commonly found in laptops and mobile phones. The other variety is the GPU card that fits onto the motherboard of a host CPU or as a separate acceleration card. Certain operations that are well suited for the GPU are then offloaded to the GPU via interconnect. It is precisely this interconnect that is a limiting factor in the use of GPUs for acceleration. The interconnect between Host and GPU using a traditional PCIe link is currently orders of magnitude smaller than the bandwidth between CPU and memory. The overhead incurred by transferring over this link can be larger than the gain in execution time.

There are solutions to the transfer overhead problem, e.g. adding more devices and/or links to split the transfer. An alternative approach is compressing the data before sending it to the accelerator. These are not solutions that work in all cases, therefore GPU manufacturers are searching for alternative methods of achieving higher bandwidth interconnect as mentioned above.

2.4. FPGA

The FPGA is an integrated circuit that does not have a defined architecture. The functionality can be configured to meet the requirements of the user. Traditionally the configuration of an FPGA was done through **Register-Transfer Level** (RTL) descriptions and converting the HDL to RTL using synthesis tools. The synthesis and routing tools are provided by the FPGA manufacturer because each FPGA type will create a different RTL implementation and bitstream. This programming file configures the logic and routing on the FPGA to match the functionality of the RTL description. An FPGA consists of several parts that will briefly be explained in the following sections. These parts are connected by programmable logic. In Figure 2.2 a basic FPGA structure is seen. As reference FPGA, the total resources available on an Arria10 GX 1150 are given in Table 2.1.

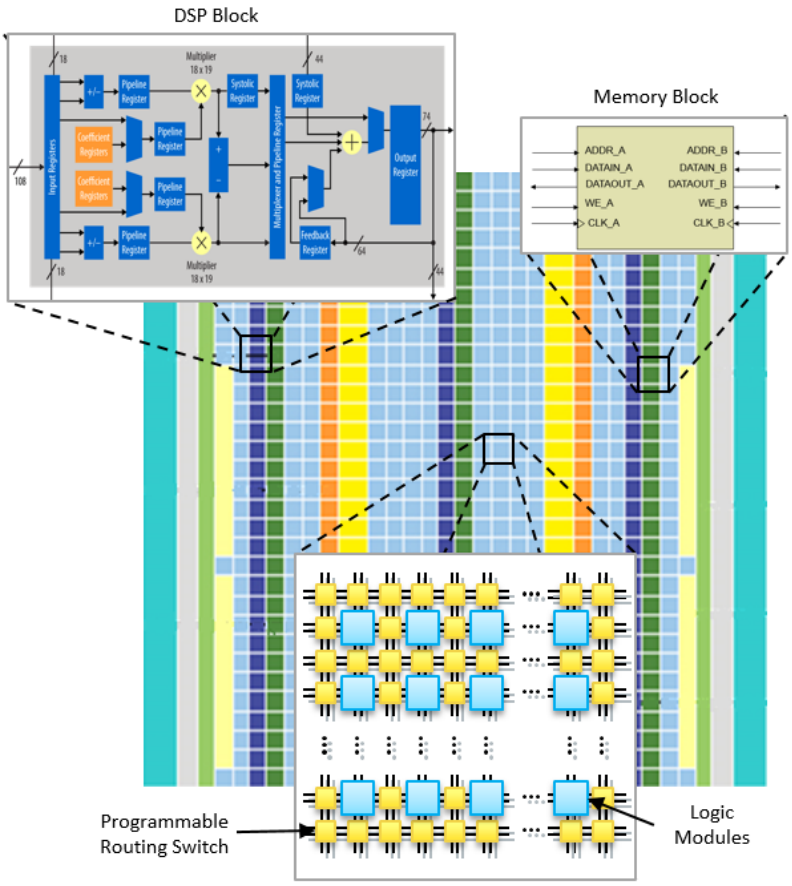

Figure 2.2: Arria10 structure with DSP, Memory and CLB highlighted [21].

2.4.1. CLB

A **Configurable Logic Block** (CLB) (also called **Adaptive Logic Module** (ALM) for Intel FPGAs) is a basic building block that creates the logic operations and local storage that make up a large part of the FPGA. They can contain **Look-Up Table**s (LUTs), Adders and registers. The contents of a CLB depend on the manufacturer and product line. The Intel Arria10 FPGA [21] contains eight inputs for the LUTs, four programmable registers and two **Full-Adder**s (FAs), the layout can be seen in Figure 2.3. The logic operations in these blocks mainly concern fixed-point or integer operations.

2.4.2. DSP

To improve the throughput and latency of variable-precision operations in modern FPGA architectures there are dedicated **Digital Signal Processor** (DSP) blocks available next to the programmable logic to reduce the number of CLBs needed for DSP suitable operations. The latest product families from

Table 2.1: Resources available on the Intel Arria10 GX 1150 [20]

Resources	Available
Logic elements (K)	1,150
System logic elements (K)	1,506
ALMs	427,200
Registers	1,708,800
M20K blocks	2,713
M20K memory (Mb)	53
MLAB memory (Mb)	12.7
DSPs (floating-point)	1,518
DSPs 18x19 (fixed-point)	3,036
Peak fixed-point performance (GMACS)	3,340
Peak floating-point performance GFLOPS/s	1,366

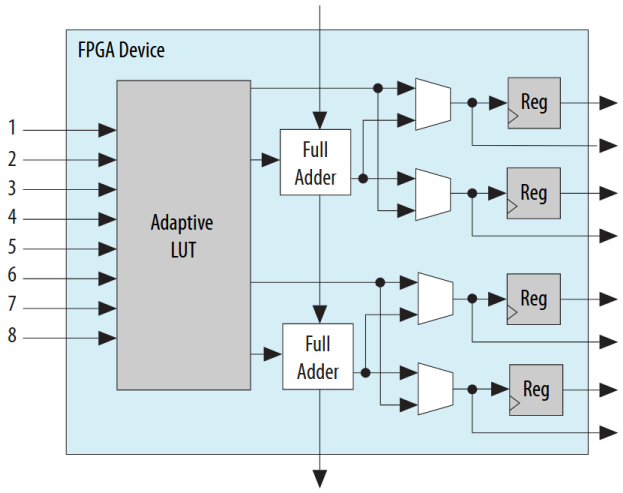

Figure 2.3: Arria10 ALM structure [21].

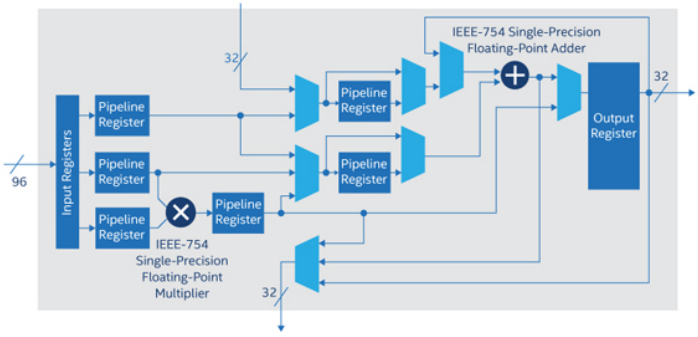

Figure 2.4: Intel Stratix 10 DSP unit in single precision mode [24].

Intel and Xilinx support single precision floating-point operations [18]. There is currently no hardened double precision support, multiplication is implemented using four DSP blocks instead of one and a increased number of registers and ALMs. The implementation of a single precision configured DSP on an Intel Stratix 10 FPGA is shown in Figure 2.4.

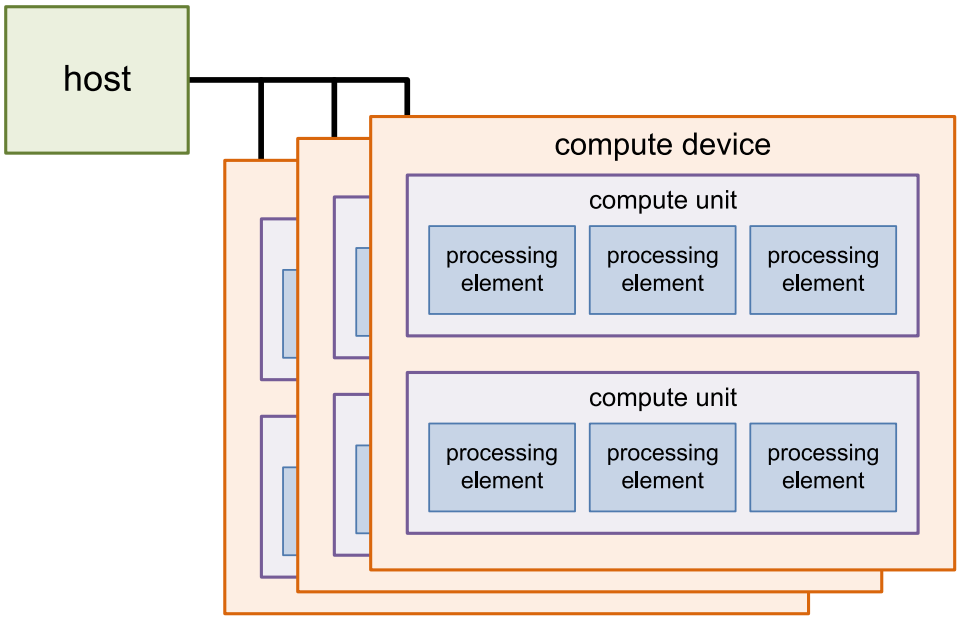

Figure 2.5: The OpenCL platform model

2.4.3. Memory

An FPGA has several types of memory. The first and smallest are registers, these are located within the CLBs. The access time of a register is a single clock cycle and the storage capacity is 1 bit. Modern FPGAs have millions of **Flip-Flop**s (FFs), i.g. 1.7 million are available on an Arria10 as seen in Table 2.1. The total storage size is still only in the order of hundreds of kBs. These registers are used as buffers between operations and pipeline stages. Larger local memory on FPGAs consists of CLBs configured as memory or **Block RAM** (BRAM), usually called M20K blocks that store 20 kilo bits. The total storage capacity is in the order of tens of Mbits, in the Arria10 only 53 Mbits of M20K memory is available. The advantage of registers compared to BRAM is that the latency is lower and every bit has an output port. BRAM only provides a few read/write ports. FPGA boards can also have external memory like **Double Data Rate** (DDR) or **High Bandwidth Memory** (HBM), this increases the total storage to GBs but has a 150x longer latency and uses resources on the FPGA.

2.4.4. Partial reconfiguration

The PR feature allows for flexibility while programming the device, while normally the whole FPGA has to be configured. PR allows only part of the FPGA to be configured while leaving the rest of the configuration intact. This allows small parts to be exchanged thus reducing the time needed for reconfiguration.

2.5. OpenCL

The OpenCL framework allows the user to execute programs across multiple devices using the same code. It is written in C with has bindings for C++ and provides an API to control and execute on the devices. Supported devices include CPU, GPU and FPGA.

2.5.1. Platform model

The model consists of a host machine that has one or more OpenCL compatible devices connected through interconnect. As seen in Figure 2.5 each compute device has a specified number of CUs which made up of **Processing Element**s (PEs). All PEs within a CU execute a stream of instructions in a **Single Instruction Multiple Data** (SIMD) or **Single Program Multiple Data** (SPMD) fashion. The sequence executed by a CU is called a work-group and the set of instructions that are in turn sent to a specific PE is called a work-item.

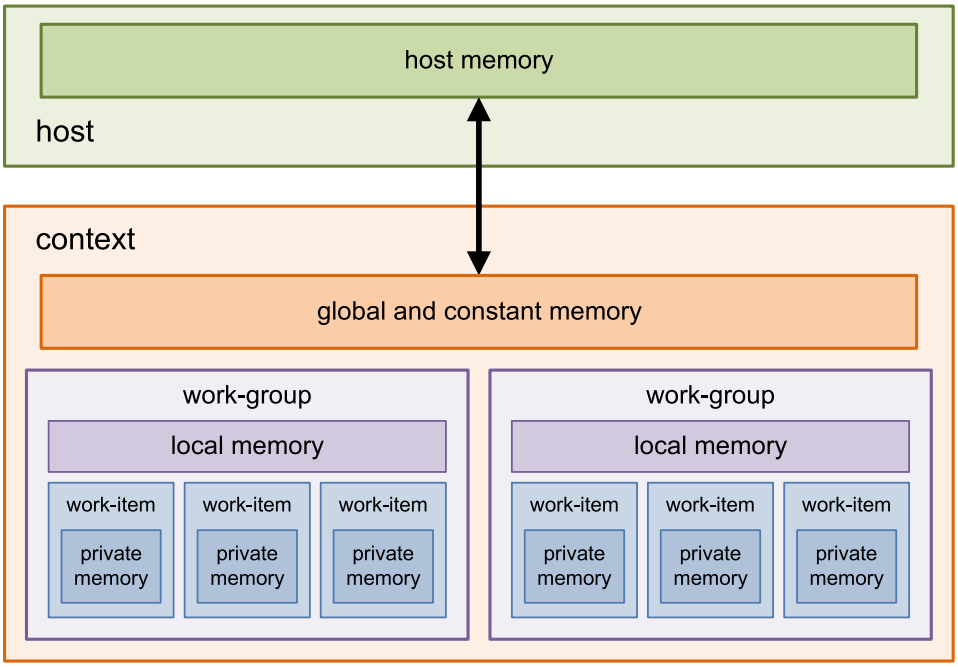

Figure 2.6: The OpenCL memory model

2.5.2. Memory model

All memory operations in OpenCL are explicit but do abstract away from manually transferring data. In Figure 2.6 an overview of the memory model is shown. The host program explicitly has to copy buffers from the host to a device, seen as context in this image. There it goes to global memory which is accessible by all work-groups. When a GPU or FPGA is used, this global memory is generally some form of DDR or HBM with a much higher bandwidth than the interconnect between host and device. Each work-group has local memory, shared between all work-items and each work-item has its own private memory for intermediate results. When the kernels have finished running, the host has to explicitly copy the result back from context global memory to the host.

2.5.3. Execution model

OpenCL programs executions are always divided in at least two parts, the host program and the kernel(s). The host program manages of the device, transports the data and controls the execution of all enqueued kernels. Each kernel instance is called a work-item, which are part of work-groups. Work-items in a group will execute concurrently on the PEs of a CU.

2.5.4. Programming model

There are two programming models that are used by OpenCL, data parallel and task parallel. In the data parallel model a sequence of instruction is executed on multiple elements of a memory object because in that case the operations are independent. The data parallel model is what people think of when GPU accelerators are considered because of the high number of cores to execute the same instruction on different data. For the task parallel programming model, tasks or kernels are completely independent, this means they can be computed out of order and even on a different devices.

2.5.5. Parallel programming

It is also possible to use HLS tools in combination standard parallel programming techniques such as OpenMP and Pthreads [5]. A logical choice to further extend ease of programming is the OpenCL framework. Because of the portability and functionality, both Xilinx and Intel (previously Altera) have added OpenCL support to their acceleration platforms. Xilinx has *SDAccel* [57], an integrated development environment for targeting their Accelerator cards. Intel has the *Intel FPGA SDK for OpenCL* [19] which supports all the latest Intel FPGAs. Both tools have already been used for various purposes, there has been great interest in FPGA acceleration from the financial and database markets. Mainly

because of their high throughput and low energy consumption [49] [53] [52].

2.6. OpenCL SDK for FPGA

OpenCL SDK for FPGA by Intel is a development environment that enables software developers to target heterogeneous platforms to accelerate their applications. It uses the programming and execution models provided by the OpenCL standard to create an LLVM project based intermediate representation of kernels which in turn is converted into Verilog which is then synthesized to a specific FPGA. The SDK provides the user with host-side and run-time environment in combination with an API to use the communicate with and transfer data to the FPGA as an accelerator.

2.6.1. BSP

Each acceleration board that uses an FPGA by Intel that supports OpenCL needs a **Board Support Package** (BSP). This is provided by the board manufacturer. The BSP is the basis of running OpenCL, it provides the environment of communication with the outside and control logic to launch and program kernels. The BSP provides all memory and communication controllers between kernels, the host and on board memory. The BSP does take up a static portion of the available resources of an FPGA, this depends on how many features the BSP provides. If the board has network ports the BSP is responsible to provide an interface between the OpenCL kernels and the physical connection. In Figure 2.7 we can see the division between user provided kernels and the BSP, everything in light blue is part of the kernel implementation. The darker blue is the BSP that communicates with the kernels, host and on board memory.

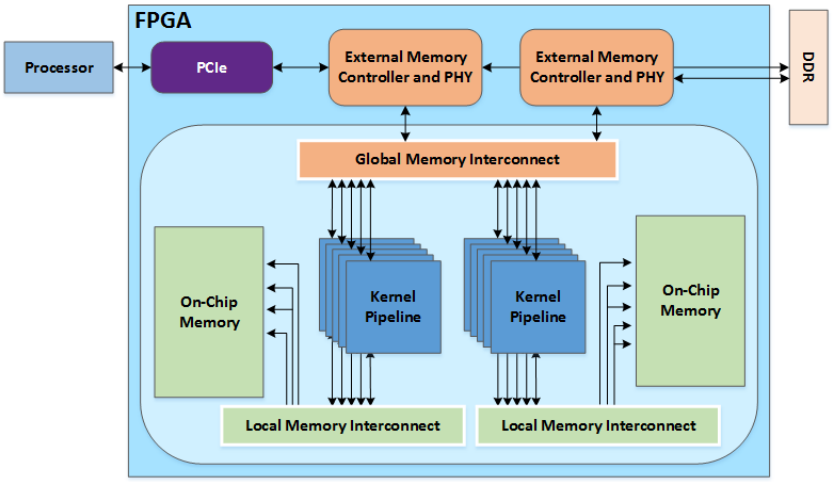

Figure 2.7: OpenCL design components, user-defined kernel pipelines in the middle, and the BSP provides the external memory controllers [22]

2.6.2. OpenCL SDK flow

Like all FPGA development, run-time compilation of bitstreams is not possible because of the synthesis and place and route effort needed. The OpenCL code in a ".cl" kernel has to be compiled to an LLVM intermediate representation, which is then converted to Verilog. From which the normal HLS flow is followed to create an FPGA bitstream (.aocx). The FPGA is programmed through the host program, when a function call is made to the kernel within that bitstream.

2.6.3. NDRange programming model

The NDRange programming model divides the work into work-items belonging to work-groups. For example, with vector addition each output vector index will be a work-item. There are several ways to get performance from the NDRange model on FPGA. Because each work-item is independent of other items. A pipeline can be created where depending on the **Initiation Interval** (II) the pipeline can be started for each work-item. It is also possible to use SIMD operations, this means widening

the pipeline i.e. adding identical PE for each stage. This uses more logic and memory but requires the same amount of control. Another option is to use more CUs and give each CU work-items. This method of partitioning parallelism increases both the control and arithmetic resource usage.

2.6.4. Single Work-Item

The other programming model available in the SDK is the single work-item. In this model the entire kernel is executed by a single NDRange work-item. The way performance is obtained is unrolling and pipelining loops. A deep pipeline with an II of one, means that each clock cycle a new iteration of the loop is started, thus optimizing the throughput. The effect of this pipeline parallelism is demonstrated in Figure 2.8. It demonstrates how each stage introduces operational parallelism, in this example 2 CUs are required to reduce the II from 4 to 1 clock cycle. Unrolling the loop around acc compresses the two **Multiply-ACcumulate** (MAC) operations to a single cycle if they are independent. This does not reduce the II rather it reduces the latency of an iteration from 4 to 3 in the example shown in Figure 2.8. In single work-item kernels, if there are dependencies across iterations of a loop, the pipeline will either have to stall or execute sequentially. The compiler does this in order to avoid these executing the operations out-of-order, at the cost of reduced performance.

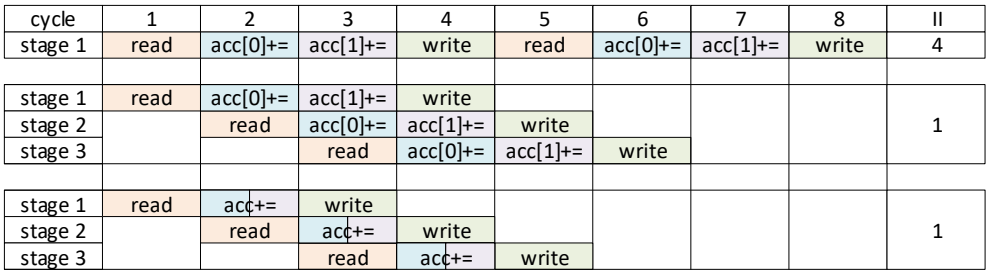

Figure 2.8: Instruction scheduling for two different pipeline depths, one version unrolls the acc operation computing both [0] and [1] in one clock cycle

2.6.5. Memory

When OpenCL **global** memory is used, the BSP will communicate with the external memory on the board which in our case is DDR3 memory and could be HBM on a high-end board. When caching is enabled in the kernel, then this will generally be implemented as private memory in **First In First Out**s (FIFOs) or registers to temporarily store loaded data. This private memory is not accessible by other PEs in the CU. When **local** memory is used it can either use registers or BRAM depending on the size and access pattern. Constant memory is implemented using BRAM with a fixed size, controlled with flags on compilation.

2.6.6. Channels and pipes

A feature specific to the Intel OpenCL FPGA toolkit is the addition of channels. A channel allows for inter-kernel, kernel-host and kernel-IO communication. Channels provides on-chip high bandwidth low latency interconnect between kernels. This allows users to only use global-memory for reading input and writing output to avoid spending time waiting on the memory interface. A channel is implemented as a FIFO buffer between kernels, the depth of this buffer can be specified by the user. This allows one kernel to push data in a streaming fashion to a different kernel, even if the other channel is not consuming it right away. If the kernel receiving the data is an autorun kernel, it does not have to be scheduled by the host program. It will automatically read from channel and execute its functionality. This saves host-device communication time, as no input from the host is needed to kick-start the computation. Traditional OpenCL pipes are supported and result in the same hardware, but not advised in the case where compatibility between SDKs is not a requirement as the OpenCL 2.0 specifications require a write must always occur before a read which conflicts with the concurrent behavior of the FPGA.

2.6.7. Programmability

One of the main purposes of adding an even higher abstraction layer on top of HLS was to bring FPGAs even closer to software programmers [30]. The OpenCL SDK for FPGAs most prominent feature is the deeply pipelined architecture created from the C-code with OpenCL directives. A new feature added by Intel is the concept of channels, as described above. This is a big improvement over using global memory to communicate between kernels [54]. As noted in [6] there are still challenges for OpenCL to overcome when it comes to programmability. A C-code based application has to be adapted for compatibility with OpenCL. This requires defining clear interfaces to global memory and implementing compute functions as separate kernels. The first RTL level synthesis report will give hints where the design can be improved by pointing out bottlenecks. With knowledge about optimizing the OpenCL code specifically for FPGAs, the user will have to optimize the design by adding pragmas to unroll loops and managing memory accesses [63]. This means using the portability of OpenCL will not automatically give acceptable performance on FPGA. The OpenCL framework is designed to divide work across CUs, not to implement complex control flows of managing hardware kernels. Advanced usage is possible, i.e. manual partitioning of data in local memory or inferring shift-registers is possible, but requires in-depth knowledge about FPGA design, which is not something all software developers have. To take full advantage of an FPGA it is often required to move to fixed-point data representations. Fixed-point is possible in the OpenCL framework but still requires manually converting between different lengths. The compiler will when signals are always zero and not implement the logic, so it is not real arbitrary precision. Another feature of the SDK is an OpenCL library, which can be created from OpenCL or RTL. That way, programmers can make function calls to optimized IP-blocks created by hardware engineers, thus using custom RTL while using OpenCL as API.

2.7. Linear algebra in qpAS

The algorithm of our industrial use-case is a linear algebra optimization problem. In this section the necessary background about the linear algebra operations is given. First, the routines from the BLAS library are explained. Second, the routines in the qpAS algorithm from the **Linear Algebra PACKage** (LAPACK) library are introduced. Third, we use arithmetic intensity as floating-point operations divided by the number of loads and stores using global memory. The arithmetic intensity is a way to provide information about the ratio between work and transfers [56]. For each routine used in the algorithm, the arithmetic intensity is calculated to see where adding additional compute units or a larger memory link could improve execution time.

2.7.1. BLAS

BLAS is not a library itself, only a specification aimed to provide portability between platforms and implementations. Originally it was developed in FORTRAN, this was the programming standard for numerical and scientific computations. The reference BLAS library is still being developed. While FORTRAN is still the default language, there is increased compatibility and integration into C and C++ languages as this is a more generally used standard. The BLAS specification prescribes a set of routines that together form the backbone of all linear algebra operations. They are divided into three levels, based on their computational complexity.

Level 1: Operations that have a complexity following $\mathcal{O}(n)$. One widely used routine is the dot product. This calculated the sum of a point-wise multiplication between two vectors. In Equation 2.1 a dot product is computed where a and b are vectors in $\mathbb{R}^n$. When double precision floating-point data type is used the arithmetic intensity of the dot product is: $\frac{2n-1}{(2n+1)8}$ [17].

$$\mathrm{a} \cdot \mathrm{b} = \mathrm{a}\mathrm{b}^{\mathsf{T}} \tag{2.1}$$

Level 2: These operations have a complexity scaling with $\mathcal{O}(n^2)$. **GEneral Matrix-Vector product** (GEMV) is one of the most used BLAS level 2 operations. In Equation 2.2 where matrix $\mathrm{A} \in \mathbb{R}^{m \times n}$ is multiplied with vector $\mathrm{x} \in \mathbb{R}^n$ the specification is given as prescribed by BLAS. The arithmetic intensity of GEMV when α and β are 1 and 0 respectively (as is commonly used): $\frac{2mn-m}{8(mn+2n)}$.

Another BLAS level 2 routine used in our algorithm solves a system of linear equations using an upper or lower triangular matrix. In BLAS level 2, TRSM solves this system for a single output vector. There is also a routine to solve for multiple vectors in level 3. In Equation 2.3 a system is solved with $A \in$

$\mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$. It scales according to $\mathcal{O}(n^2)$ but is data dependent which makes parallel execution more difficult. The arithmetic intensity of a forward or backward substitution (the naive algorithm used) is: $\frac{n^2}{4(n^2+5n)}$

$$\mathrm{y} \leftarrow \alpha\mathrm{Ax} + \beta\mathrm{y} \tag{2.2}$$

$$\mathrm{Ax} = \alpha\mathrm{b} \text{ solve for } \mathbf{x} \tag{2.3}$$

Level 3: Is the highest level of BLAS, operations scale with the computational complexity of $\mathcal{O}(n^3)$. **GEneral Matrix-Matrix product** (GEMM) is the best known BLAS operation in this category where two matrices $\mathrm{A} \in \mathbb{R}^{m\times k}$ is multiplied with $\mathrm{B} \in \mathbb{R}^{k\times n}$ resulting in matrix $\mathrm{C} \in \mathbb{R}^{m\times n}$. The general version as provided by the BLAS specification is given in Equation 2.4. In the general case α and β are taken to be 1 and 0 respectively. In that case the arithmetic intensity of GEMM using double precision is as follows: $\frac{2mnk-mn}{8(k(m+n)+mn)}$.

$$\mathrm{C} \leftarrow \alpha\mathrm{AB} + \beta\mathrm{C} \tag{2.4}$$

2.7.2. LAPACK

Where BLAS specifies basic operations LAPACK contains more complex routines for solving systems of linear equations, least-squares solutions, eigenvalue problems, singular value problems and multiple methods of matrix factorization. The routines in LAPACK call BLAS functions for the basic operations that are used. Taking advantage of SIMD in BLAS is straightforward as the output elements are calculated independently. This is not the case in LAPACK, there most routines are sequential in nature because of the data dependencies in the algorithms.

The two main LAPACK routines used in the qpAS algorithm are TRTRS and POTRF. Where TRTRS is a triangular solver that is just a wrapper around the BLAS routine TRSM or TRSV depending on whether it solvers one or more vectors, the LAPACK call adds singularity checks. POTRF is the subroutine that computes the Cholesky factorization of a matrix A as given in Equation 2.5. There are two options for factorization, a lower triangle or an upper triangle. In Equation 2.5 the factorization is used to compute the lower triangular L of A. Both A and L are square matrices with $\mathrm{A}, \mathrm{L} \in \mathbb{R}^{n\times n}$. Cholesky factorization only works for input matrices that are real and **Symmetric Positive Definite** (SPD). The arithmetic intensity of the factorization is: $\frac{n^3}{12(n^2+n)}$

$$\mathrm{A} = \mathrm{LL}^{\mathrm{T}} \text{ calculate } \mathrm{L} \tag{2.5}$$

3

Related work

In this chapter, we present the current state-of-the-art solutions and research relevant for our work. First, we will explore the differences between FPGA and GPU that are related to linear algebra and floating-point performance. Because of the scope, we focus on the use of FPGA accelerators for offloading various applications. Second, we will explore what work has already been done on the topic of offloading HPC tasks to the FPGA, with and without a high-level abstraction. Third, the current state-of-the-art implementations on the FPGA to compute routines in the BLAS and LAPACK library are evaluated to see what we can build on in our research and what is not yet present.

3.1. FPGA vs GPU

The main difference between the GPU and FPGA is of that FPGAs are dynamic and GPUs are static. Where an FPGA can adapt the number of cores and capabilities of a core to the application a GPU has a static number of cores which far outnumbers the cores possible on an FPGA. GPUs were increasingly adopted in the last 10 years by HPC application programmers because of the need for more memory bandwidth and higher FLOPS/s [40]. It has already been a topic of interest for longer, but it became increasingly feasible by improvements in the GPU architecture. In current architectures, GPUs offer half the throughput for double precision compared to single precision. For FPGAs it has mostly been fixed-point precision signal processing [10] where speedups of tens to hundreds were achieved between CPU and FPGA. This depends on the algorithm and dramatic improvements are seen for compute bound algorithms compared to low speedups for I/O bound. Another field where FPGAs are common is control based algorithms where deterministic behavior is a requirement [60]. They present a controller that includes fixed-point arithmetic and meets their high requirements by using the parallel capabilities of an FPGA. There have been several studies comparing FPGA and GPU to CPU. Additionally, there is research into what kind of algorithms do well on which platform. In [4] they conclude that FPGAs are well suited for data streaming and pipelined algorithms or applications that require low-level operations not supported by high-level languages. They also conclude that the GPU is well equipped to handle applications with little data dependencies that can be computed in parallel. In [28] the authors present a comparison between the platforms for typical HPC routines from the BLAS library. They compare the performance of a BLAS level 2 routine in double precision floating-point. Where FPGA is the most energy efficient, but achieves lower performance in terms of execution time than the CPU. The input data sizes are relatively small to properly take advantage of the GPU. Since then, manufacturers of GPUs have improved the floating-point performance. We expect these results to look significantly different taking this into account. The common trend among research is that when the work-load or data-size is large enough and the operations use floating-point data, GPUs with their high number of cores will outperform the other platforms. Given that the problem can be split in a large number of work-items to be divided among all processing units. Where the GPU wins in terms of throughput, an FPGA can win in energy efficiency.

3.2. OpenCL on FPGA

The concept of HLS is old, but the use has become increasingly popular [39]. HLS was proposed to speed-up development by using high-level languages to generate RTL. But it is still made for programmers with intricate knowledge about the underlying fabric. In the following sections we discuss the performance of HLS based implementations and alternative solutions.

3.2.1. Performance

There have been various studies into the performance of OpenCL SDK for FPGA using HPC applications and benchmarks. One example of such a study explores the performance of the Smith-Waterman algorithm on FPGA using the OpenCL SDK from Intel [16]. Their implementation does not use central control, instead there is explicit synchronization by using autorun kernels. This allows kernels to work independently. Their PEs achieve high utilization by using the channel concept introduced by Intel. In HPC the most common data-types used are from the IEEE-754 [18] floating-point standard. For this reason both Intel and Xilinx have added hardened floating-point support to their latest FPGAs. This only supports single precision, double precision is still not fully supported. In [26] the authors found that there is still a high penalty in performance using double precision on FPGA. The benchmark used was more than five times slower on FPGA than on CPU. For single precision, the performance was only two to three times slower. In [31] the authors explore the performance of single precision floating-point operations using the hardened support provided by the DSPs present on the Arria10 product family. One of the benchmarks they use is a matrix multiplication using OpenCL. Using 82.5% of the DSPs, they manage to get a around 730 GFLOPS/s. The authors of [25] found that the performance of an application that simulates a nuclear reactor, which is a floating-point intensive kernel was 4.5x and 6.4x lower than GPU and CPU, respectively. The conclusion they draw is that when energy consumption is a metric, using the FPGA can be beneficial. There are applications where FPGA can deliver competitive or better performance than a GPU and CPU. In [48] the authors compare an OpenCL FPGA implementation of a 3D FFT to a CPU, a GPU and an FPGA implementation. The OpenCL-HDL implementation has a faster execution time for smaller work sizes (FFT window size of 16^3, 32^3 and 64^3). These window sizes might work well for FPGA, but are small enough that efficiently using the massively parallel GPU will be difficult. There is already less difference between GPU and FPGA when going from size 32 to 64, the expectation is that when sizes grow the GPU will beat the FPGA.

Intel is not the only one with an OpenCL FPGA interface to use heterogeneous hardware platforms. The pocl (Portable OpenCL) project allows for portability between devices. In [15] the pocl framework is used as an interface to program a VLIW architecture. The authors add their optimized cores to the framework, allowing developers to use those cores transparently to optimize the execution time of their application with minimal effort.

3.2.2. Alternatives

Directly writing OpenCL kernels is not the only option, as shown in [34], where the authors create a framework to translate C-code with OpenACC [44] directives to OpenCL. This code can then be used in the Intel SDK for FPGA. This decreases the effort needed by programmers to port their application to OpenCL. In the generated OpenCL implementation, optimizations are still needed to get decent performance. There are also options that do not use OpenCL as an intermediate step. In [3] a hybrid solution is generated where a soft processor works together with a HLS based accelerator. There have been studies on the performance of LegUp compared to other c-to-vhdl compilers [39], and to the Intel OpenCL SDK for FPGA [46]. The conclusion of these studies is that the Intel OpenCL SDK optimized implementation performs the best for the chosen applications. An often used approach in image processing applications is using custom softcore processors to perform signal processing applications [14]. In this paper, streaming data between cores and from the memory interface is used to get efficient compute cores that handle SIMD based program flows. Kernel development is not the only aspect involved when moving to an accelerator. If multiple data representations and languages are used serialization of data is often required which reduces the benefit of using an accelerator. Frameworks such as Fletcher add the functionality from Apache Arrow to efficiently stream data to an accelerator with less effort [45].

3.3. BLAS and LAPACK on FPGA

A requirement in this work is to use COTS components, the same holds for the computational libraries. For this reason the BLAS and LAPACK are chosen. Several studies into the performance of offloading BLAS and LAPACK routines to FPGA exist. In this section we try to determine the potential of FPGA in this context, and identify the gaps.

3.3.1. BLAS

The study presented in [28] was one of the first to compare performance of the BLAS routine *gaxpy* on different platforms. *gaxpy* is a BLAS level-2 routine, of which the compute complexity is $\mathcal{O}(n^2)$. As GPUs specialize in exploiting massive parallelism they only outperform the other platforms for a large enough problem sizes. This study predates the hardened single precision support in the FPGA DSP blocks. For the BLAS level-3 routine SGEMM, this support increased single precision GFLOPS/s performance by 3.9x from the Stratix V to an Arria10 [51]. In [47], an FPGA-based accelerator matrix multiplication is presented. While their method is old, it highlights that level-3 operations benefit more from acceleration than level 2 and 1 because of the **instruction-level parallelism** (ILP) and **data-level parallelism** (DLP). Even though there was no hardened single precision support at that time, double precision GEMM on FPGA still achieved 60% more performance compared to a high-end CPU at that time.

In the last decade GPUs, with their massive parallelism are the go-to for ILP and DLP applications [40]. This is also the case for double precision data types, in recent architectures NVIDIA and AMD both achieve a double precision GFLOPS/s throughput of 50% compared to single precision. There are operations for which FPGAs are competitive. In [62] the authors present an image recognition algorithm where the FPGA achieves similar execution time and outperforms the GPU in terms of energy efficiency. In [48] the authors present an FPGA implementation of a 3D-FFT that is faster than the CPU and GPU for their chosen sizes. Other Algorithms that fit FPGAs well are sequential and have dependent operations such as FIR-filters [31].

While there are no fully capable OpenCL BLAS or LAPACK libraries available, there is continuous work creating them. In [35] the authors present a small subset of the BLAS library that is compatible with the Intel OpenCL capable FPGAs. They use tiling and channels to create parallel streamed processing elements. Their results show that the CPU using COTS libraries is still faster in terms of execution time but the FPGA is more power efficient.

A common trend in the mentioned studies is that they highlight the energy efficiency of the FPGA compared to the CPU and GPU. In many of the studies the FPGA can be more energy efficient at the cost of execution time or throughput. Most of the previous studies explore the performance of one or more routines, but there few studies that evaluate a complete algorithm in the domain of linear algebra. The difference between comparing single routine execution to a complete algorithm is the use of data streaming between compute kernels. Throughput in terms of GFLOPS/s is not the best indicator of performance. In our case latency is more important than throughput. Each call of the algorithm computes corrections to improve machine performance. The worst-case execution time is critical as there is a strict deadline on the input of the corrections.

3.3.2. LAPACK

There have been several studies that research the offloading of LAPACK routines to hardware accelerators. In [13] the GPU performance of the Cholesky decomposition is compared to the CPU performance for both single precision and double precision. For small matrix sizes, the CPU performance is still competitive. When the size grows the GPU will outperform the CPU. Even when the transfer of data is included in the comparison. In 2010 Yang compared the performance of then state-of-the-art FPGA, GPU and CPU implementations [61]. In 2010 GPUs where just starting to gain traction as accelerators and still performs better than both CPU and FPGA. A more recent paper by Intel has shown significant improvement for the QR-decomposition [32]. Langhammer and Pasca achieved 744.2 GFLOPS/s for a 512 sized single precision input matrix. This implementation efficiently uses the hardened floating-point support of the Intel FPGA families. When double precision is used, we expect at least a factor 4 lower performance as these operations use 4x as many DSPs in addition to increased ALM and register usage. These numbers are still no match for a comparable GPU implementation [13] and for small sizes on a CPU. These studies show that the FPGA will not win in terms of throughput for single routines,

but might for a complete algorithm if the features of the FPGA can be used correctly.

4

Quadratic programming Active-Set

In this chapter the qpAS algorithm used in our use-case is explained. The most important metric in our case it is the execution time of the algorithm. To get a better idea which parts have the most influence on the execution, we profile the reference CPU implementation. With the critical parts identified, we try to understand which properties cause it to scale in complexity leading to a long execution time. These characteristics of the algorithm can then be used to get an idea about what options there are to speed up the algorithm. Although we include the three major hardware platforms (the CPU, GPU and FPGA), the implementation will be an OpenCL based FPGA implementation in accordance with the requirements.

First, we start with an explanation about the set of algorithms the qpAS algorithm belongs to, as it is an optimization problem. Second, we profile the implementation as presented in the use-case, which uses Intel **Math Kernel Library** (MKL) routines on CPU. Third, the mathematical and computational characteristics are discussed, each with the hardware deployment in mind. Fourth, Each of the hardware platforms is discussed as a candidate to deploy parts of the algorithm or the complete algorithm. Last, we talk about the bottlenecks our implementation focuses to solve using the OpenCL abstraction on the FPGA.

4.1. Optimization problems

The quadratic programming Active-Set algorithm is a special case of quadratic programming which is a set of programs that solve quadratic problems. The algorithm tries to minimize the input to find a solution that satisfies the constraints. The quadratic problem is given in Equation 4.1 [42]. Where G is a symmetric $n \times n$ matrix, ε and I are finite set of indices and c,x and $a_i, i \in \varepsilon \cup I$ are vectors in $\mathbb{R}^n$.

$$\begin{aligned} \min_x \quad & q(x) = \frac{1}{2}x^T G x + x^T c \\ \text{subject to} \quad & a_i^T x = b_i, \qquad i \in \varepsilon \\ & a_i^T x \geq b_i, \qquad i \in I \end{aligned} \tag{4.1}$$

Our algorithm is an adapted version of the general case, where we have upper boundary and lower boundary constraints. The pseudocode of this algorithm is shown in Algorithm 1. In the next sections we will discuss how the algorithm works and what is important to focus on while profiling and we define the critical path.

qpAS is an iterative optimization method, in our case we have to minimize Ax within the provided boundary conditions.

$$\min_x \frac{1}{2}x^T H x + f^T x \quad \text{subject to} \quad lb_A \leq Ax \leq ubA \tag{4.2}$$

In Equation 4.2 the mathematical representation of the problem is given, with $H \in \mathbb{R}^{n \times n}$ (SPD dense matrix). Vector $f \in \mathbb{R}^n$ and matrix $A \in \mathbb{R}^{m \times n}$ model the constrains, where n is the size of the original Hessian matrix H. Vectors lb_A and ub_A represent the lower and upper bounds, both of them have m elements. The algorithm aims to find a vector $x \in \mathbb{R}^n$ that satisfies the requirements in Equation 4.2.

Algorithm 1: A high-level overview of the qpAS algorithm.

Input: $H \in \mathbb{R}^{n \times n}; A \in \mathbb{R}^{m \times n}; f, x_0 \in \mathbb{R}^n; ub, lb \in \mathbb{R}^m$
Result: x
x_0 feasible;
$ax := Ax_0$;
for $i = 0, \ldots, n-1$ **do**
  if $\left|ub_i - (ax)_i\right| < \epsilon$ **then**
    $W_i = 1$;
  else if $\left|lb_i - (ax)_i\right| < \epsilon$ **then**
    $W_i = -1$;
  else
    $W_i = 0$;
  iter = 0;
while *iter* < *maxiter* **do**
  solve $\begin{pmatrix} H & A_{as}^T \\ A_{as} & 0 \end{pmatrix} \begin{pmatrix} x_{new} \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ b_{as} \end{pmatrix}$;
  $\Delta x = x_{new} - x_{old}$;
  if $(\|\Delta x\| < tol)$ **then**
    $\lambda_u = \lambda_{1:n_u}$;
    $\lambda_l = -\lambda_{n_u+1:end}$;
    if $\lambda_u \geq 0$ *and* $\lambda_l \geq 0$ **then**
      converged;
      $x = x_{new}$;
    else
      deactivate constraints;
  else
    find blocking constraint $adx := A\Delta x$;
    $u_i := \frac{adx_i}{\left|ub_i - (Ax_{old})_i\right|}$;
    $l_i := \frac{adx_i}{\left|(Ax_{old})_i - lb_i\right|}$;
    argmax$_{j \in \{1,\ldots,m\} \setminus AS} \{l_j, u_j\}$;
    determine the corresponding step length s;
    $x_{new} = x_{old} + s\Delta x$;
    update working set W_i including a check for linear dependency;
  iter = iter + 1;

4.1.1. Quadratic programming

Any quadratic problem can be shown to have a solution or else show the problem is infeasible. Because we know there is a solution, the computation required is finite. The effort needed to get the solution is heavily dependent on the Hessian matrix H and constraints ub_A and lb_A [42]. In our case the matrix H is SPD, this means that the problem is *strictly convex*. Having an SPD matrix allows us to use the *Cholesky factorization* instead of a more general factorization method like the *QR-factorization*. This reduces the computational complexity of the algorithm but imposes the SPD requirement on the input data.

4.1.2. Active-Set

An important aspect of the algorithm is that it works on an Active-Set. It keeps track of a working set W consisting of active constraints, which is a subset of ubA and lbA. The reason Active-Set is used is that when the problem is large, including the whole solution space requires significantly more effort. Another property of the Active-Set method is the stability, at any point during the algorithm, the intermediate solution is in the feasible region. During each iteration of the algorithm, The l^2 norm (vector norm) of the output vector dx is checked if it lies within the required tolerance. If this condition is met, there is still a possibility that the solution does not satisfies all conditions. For this we check if all Lagrangian multipliers (λ_u and λ_l) are positive. If there are still negative multipliers one of the corresponding constraints can be dropped from the Active-Set.

Tracking the working set W_i where i includes all active constraints is important to retrieve the correct vectors A_i^T that together form the active set A_{as}. The number of constraints present in the Active-Set W_i strongly impacts the execution time required for calculating the Lagrangian multipliers.

4.1.3. The critical path

In our application we consider the critical path to be the loop on line 11 in Algorithm 1. The operations needed before this loop can be precomputed to reduce their impact on the critical path. The critical path only depends on the operations within this loop. Because the goal is to meet a given deadline, we should focus on accelerating these operations. Some of the operations now present within the loop do not change across iterations. As such, they could be moved outside of the loop to further reduce the work necessary in the critical path. Such optimization are platform independent and will work for any kind of implementation. Regardless, they are still important to take into consideration when choosing a hardware platform as they impact the compute and memory requirements differently.

4.2. Profiling

To get a better understanding what parts of the algorithm impact execution time we profile the reference implementation on a CPU. As the algorithm runs for a unknown number of iterations until the solution is converges To get a clearer picture of the behavior, we look at the time each iteration takes instead of the complete algorithm. This way, we can see if there is scaling of the execution time with the number of iterations. We run the algorithm for two data sets because the number of iterations needed to find the solution depends on the start vector and the constraints. The first use-case will have a low number of maximum iterations (30) which from here on will be identified as use-case (a), the other will have a high number of maximum iterations (180) which is identified as use-case (b). These data sets are chosen because the variation in execution time of a single iteration is mainly caused by the number of elements present in the Active-Set.

Each iteration, a check is performed on the new solution which determines if it satisfies the requirements. This means that the algorithm can converge to a solution faster than the specified number of maximum iterations. This maximum exists to limit the execution time of the algorithm, to prevent exceeding the deadline. If this maximum is reached, the solution found at that iteration is used, even if the solution is not yet converged.

The CPU implementation used for profiling is C-code based, using the MKL library provided by Intel [23]. The configuration of the profiling system is outlined in Table 4.1. It uses function calls to an optimized BLAS and LAPACK library provided by Intel that contains routines for Cholesky factorization, triangular solving and matrix multiplication. To profile the general behavior and execution time of iterations, CPU timers are used. These are accurate in the order of μs which is enough for our use. For a more in depth call graph profiling of the application, the callgrind tool present in valgrind is

Table 4.1: The setup used for profiling, all cores enabled

Item	Value
CPU	Intel Xeon Gold 6226 (single socket/ 12 cores)
DRAM	384 GiB DDR4 2666 ECC
OS	RHEL 8 (64-bit)
Compiler	GCC 4.8.5
MKL	Intel MKL 2019 update 3

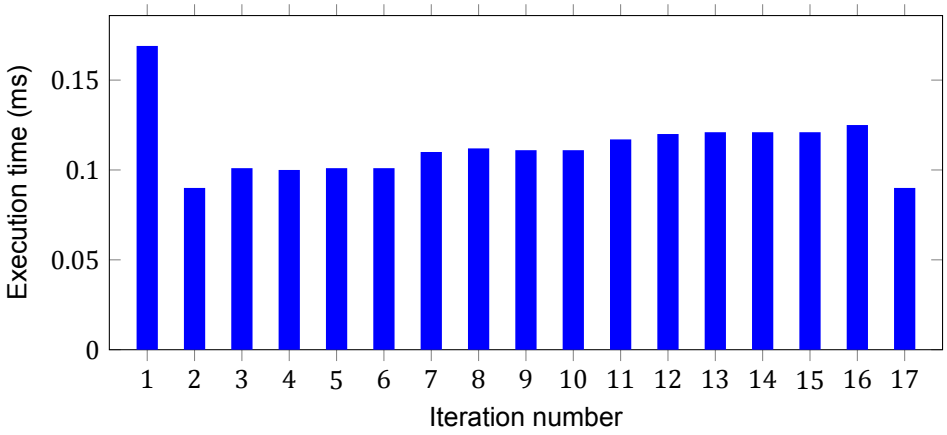

Figure 4.1: Average execution time in ms per iteration for use-case (a) on CPU

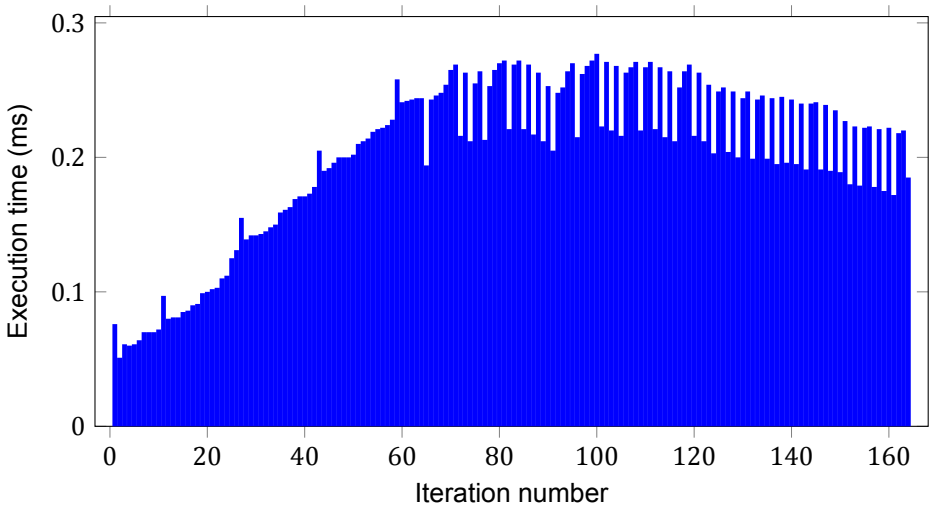

Figure 4.2: Average execution time in ms per iteration for use-case (b) on CPU

used [50]. Some optimizations discussed in the previous sections and not shown in Algorithm 1 are already present in the reference implementation.

There are small differences in input data for the two cases. Use-case (a) has a lower maximum number of iterations but has more constraints $m \approx 3000$, where use-case (b) only has $m \approx 700$. This impacts most of the control decisions and the Adx matrix-vector multiplication. Other operations such as the Cholesky factorization and the substitutions are dependent on the number of elements in the Active-Set. Size n of input matrix H is similar between the use-cases (≈ 200) and thus has no impact. In Figure 4.1 and 4.2 the execution time for both use-cases is plotted against the iteration number. It can be seen that there is a correlation between the iteration and the execution time. This is more visible for use-case (b) as it has a higher maximum number of iterations. Another thing to notice is that the first iteration takes significantly more time than any subsequent operations. This is likely caused by the

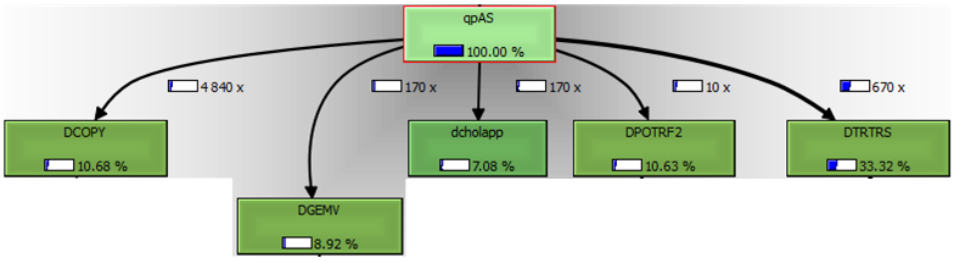

Figure 4.3: Call graph of qpAS with a low number of iterations, use-case (a)

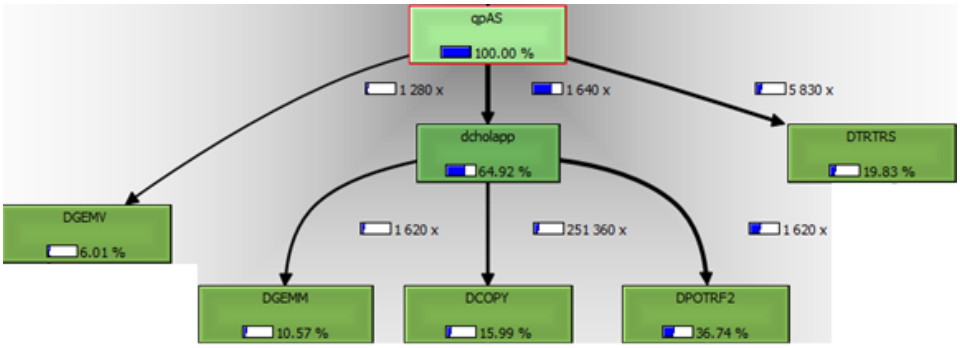

Figure 4.4: Call graph of qpAS with a high number of iterations, use-case (b)

first use of buffers and mkl library calls and computations repeat across iterations.

For use-case (b) after the 65th iteration there is a stagnation in the increasing execution time. Before this point constraints were mostly added. After, it is a combination of adding and removing constraints from the set. The execution path for adding or removing a constraint is not the same, in the case where a constraint is added, the Adx matrix-vector multiplication is needed. This operation is not required when a constraint is removed from the Active-Set, in that case the factorization is updated to subtract the influence of the constraint from the set. This causes the zigzag behavior in the execution time as skipping the matrix-vector multiplication reduces the iteration time. Some operations in the algorithm happen before the norm of dx is checked and are executed each iteration. This static part consists of the Cholesky factorization, the forward and backward substitution. The steady decline in execution time is caused by an overall decrease in the number of constraints present in the Active-Set.

To get a more in-depth explanation where this increase in execution time originates, the callgrind tool from valgrind is used [50]. The call graph of use-case (a) given in Figure 4.3 provides insight into the call behavior when only a low number iterations are performed. In this case the triangular solver (TRTRS) is called for the full size n of H multiple times. Because there are only a few constraints present in the Active-Set for this use case the *dcholapp* function causes only a small portion of the execution time. This changes for use-case (b) seen in Figure 4.4, where the dcholapp function internally does more work. This consists of matrix-matrix multiplication (GEMM) over all vectors A_W in the working-set and factorization (POTRF) of the resulting matrix. As expected, this takes increasingly more time when the number of constraints in the set grows. For both use-cases a significant portion of calls are made to copy operations, these are used to create new data structures like L and copy parts of A to A_W.

The callgrind tool only looks at the calls of functions which can give a warped view of the division in execution time. To get insights about the difference in execution time between an iteration as a whole and only the cholapp function more measurements are needed. For this we add CPU timers around the while loop to measure the time it takes to perform an iteration of the loop. Additionally, a CPU timer is added around the cholapp function to measure time spent within this call. The results of these measurements can be seen in Figure 4.5. There is a correlation between the execution time of the cholapp function and the total time spent in an iteration. The cholapp function is the largest contributor to the increase in execution time. The substitutions on the Active-Set in the rest of the iteration cause the growth in the other part.

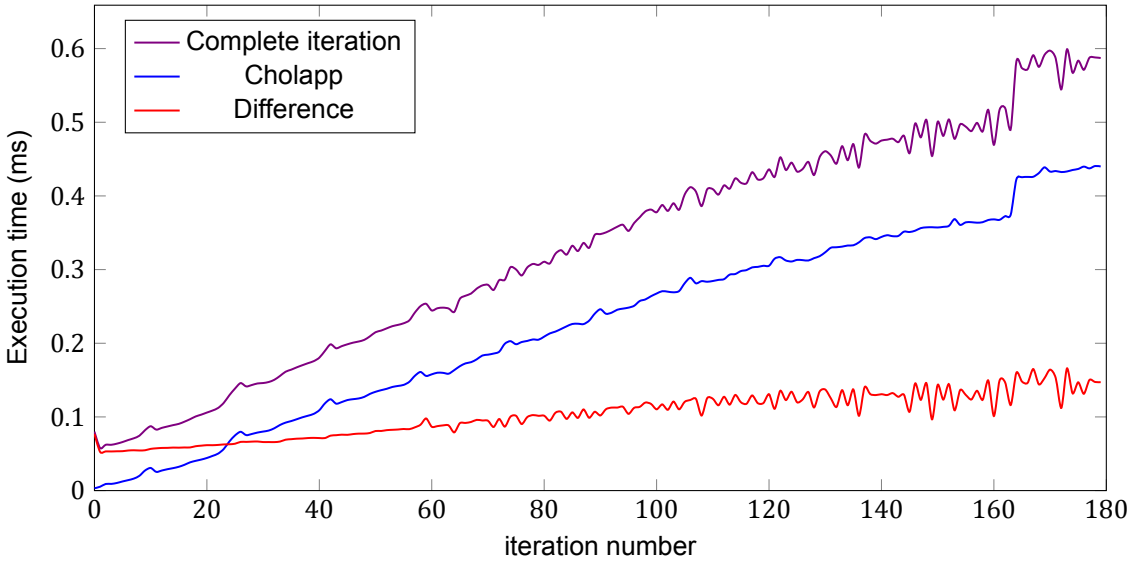

Figure 4.5: Execution time of function cholapp for each iteration with total iteration time and difference plotted

Bottlenecks In this section we talk about bottlenecks in the algorithm and reference implementation. First, we we talk about the algorithm bottlenecks. In specific, what steps are necessary to execute steps of the algorithm in parallel. After, we discuss the implementation bottlenecks that are currently present in the reference design.

The growth in execution time is mainly caused by the growth of the Active-Set. The other part, which is generally static static across iterations consists of mostly control operations which are all vector updates, comparisons and the matrix-vector multiplication. It is beneficial to also reduce the static part of the algorithm, but it will not solve the growth in execution time caused by the Active-Set. As discussed in the previous section, the function that grows fastest with the Active-Set is the cholappend. It computes the Cholesky factorization of the Active-Set with the new constraint. In the current implementation, it computes a matrix multiplication of n_{as} vectors of n size. Afterward, the resulting matrix is factorized in full, even though only one row from the input is changed. The two operations both scale with $\mathcal{O}(n_{as}^3)$ if computed in full. The forward and backward substitutions have to wait for this factorization to finish before the new x vector can be determined.

The arithmetic intensity of the Cholesky factorization is given in Equation 4.3 scales with $\mathcal{O}(n^3)$. As the Active-Set increases, the computational complexity grows more than the required memory accesses. Solutions that improve the execution time of the factorization should provide compute over bandwidth. The arithmetic intensity does not provide an insight into local memory usage. Factorization is a sequential algorithm because of data dependencies, with some intermediate steps that can be executed in parallel. Because of this data dependency, it is important have fast floating-point compute units. The matrix-multiplication does not have data dependencies and can therefore be executed completely in parallel.

The major bottlenecks from the CPU implementation to focus on are: the Cholesky factorization and the matrix-multiplications. The control operations are static across iterations and contribute little to the overall execution time if a large number of iterations are needed. In the CPU implementation the substitutions account for a small portion of the execution time, this might different on other hardware platforms as these are also sequential operations.

$$\text{Arithmetic intensity} = \frac{\frac{n^3}{3}}{\frac{8(n^2+n)}{2}} \rightarrow I = \frac{n^3}{12(n^2+n)} \tag{4.3}$$

4.3. Characteristics

From the profiling and analysis we identify four main performance characteristics of the qpAS algorithm. Each of these four aspects is discussed separately in the next sections. With these characteristics, we

aim to identify the best implementation and deployment strategy, which will help during the implementation of the algorithm on FPGA using OpenCL.

4.3.1. Dynamic ranges

Not all data structures in the algorithm remain constant in size across iterations. As this is an Active-Set algorithm, all arrays that depend on the size of the set will change by at least one column or row each iteration. The structures that have a dynamic size are: $A_{as} \in \mathbb{R}^{n_{as} \times n_{as}}$, $\lambda \in \mathbb{R}^{n_{as}}$ and $b_{as} \in \mathbb{R}^{n_{as}}$ where n_{as} is the number of constraints in the Active-Set and where n_l and n_u are the number of lower and upper bounds in the Active-Set respectively. If a naive implementation of the algorithm is executed, the linear system solver first has to factorize H with A_{as} added. The size of A_{as} is $n_a s$ vectors of size n. The total size of the factorization is then $n_{tot} = n + n_{as}$. The combined computational complexity of the factorization and solving the total system is:

$$n_{tot}^3/3 + 2(2n_{tot}^2) \tag{4.4}$$

The factorization of H is constant across iterations. The operation can be computed once outside of the loop and reused multiple times, removing it from the critical path. Using this method reduces the size of the factorization to only $n_l + n_u$. These are added to the precomputed factorization and combined with the factorization from the Active-Set to give the lower triangular matrix L [42]. This means that only $n_{as}^3/3 + 2(2n_{tot}^2)$ operations are needed compared to the total system. The complexity still scales cubic with a reduced in size of n compared to computing the complete factorization and solving which takes the number of operations mentioned in Equation 4.4. This has little impact for the initial constraints. As more constraints are added, the execution time of the algorithm is still dominated by the factorization. Further optimizations for the Cholesky factorization are possible. As only one constraint is added or removed per iteration, it is redundant to perform the complete factorization. For adding a constraint, only one new row has to be computed and added to L, if it is added at the bottom, the other rows of L stay the same. Calculating this last row still takes effort, but it scales quadratic instead of cubic. When a row is removed, only the rows beneath it need an update to subtract the effect of the deleted row.

Dealing with constantly changing data sizes is easy to manage on CPU. Loops that depend on variables are easily scheduled, and memory structures can be combined with little effort. Parallel libraries such as OpenMP or MKL are not impact in performance if the size changes each iteration. There is enough capacity on the CPU to allocate the largest possible size in memory and only use what is needed.

However, on architectures designed for SIMD operation the impact of the dynamic data structures can be severe. A GPU favors deterministic control flows. Dependent loops and changing input sizes can cause warp divergence, which in turn impacts the performance. A different problem arises when considering the FPGA architecture. On a FPGA, the buffers and local **Random-Access Memory** (RAM) blocks are statically determined at design time. Memory blocks have to be allocated large enough to suit the maximum needs of the algorithm. If the a full matrix of size $n_{tot} \times n_{tot}$ resides in local memory this will occupy a significant part of the RAM blocks. Modern FPGAs have can have in the order of hundreds Mbits of local memory. Storing input data and intermediate results local memory is preferred, as the access latency involved with accessing global memory is significantly larger. When data does not fit into the local memory structures, global memory can be used. Most FPGA accelerator boards have off-chip DDR memory or HBM available large enough to fit all of our data. Loading and saving to/from global memory causes a transfer overhead, this increased access latency can stall pipelines and slow down execution.

The chosen solution should be able to handle the largest size that is caused by the maximum number of constraints present in the Active-Set. Either in off-chip memory or in local memory, the data load/store latency will impact performance if memory access patterns are not taken care of. If global memory is used, the load/stores should be made non-blocking to the pipeline thus avoiding stalling the pipeline. Efficiently handling the changing sizes and increased complexity of the factorization and the subsequent substitutions is required. The solution needs to reduce the work by only adding the new data to both the factorization and forward substitution.

4.3.2. Sequential execution

Several parts of the algorithm are sequential because they depend on data from previous operations. In this section, the consequences of this sequential behavior on acceleration of the algorithm are dis-

cussed. We start with the sequential behavior within operations. First, Cholesky factorization is discussed and different implementations are analyzed. Second, triangular solvers and its properties for parallel execution are evaluated. Last, we look at the sequential behavior across operations and the impact thereof.

Cholesky factorization The Cholesky factorization has data dependencies that cause sequential behavior. If we look at the pseudocode provided in Algorithm 2, both L_{ij} and L_{ii} depend on previously calculated values of L.

Algorithm 2: Naive Cholesky factorization

Data: $A \in \mathbb{R}^{n \times n}$
Result: $L \in \mathbb{R}^{n \times n}$
for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $i - 1$ **do**
    $L_{ij} \leftarrow (A_{ij} - \sum_{k=0}^{j-1} L_{jk} L_{ik}) / L_{jj}$
  end
  $L_{ii} \leftarrow \sqrt{A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2}$
end

There are two main ways to execute the factorization: row based or column based. The consequences of choosing either impacts the memory access pattern. An often used version in parallel architectures is the right-looking column based algorithm. Within this category there are two versions, one where the original matrix A is updated, another is shown in Algorithm 2 and uses a running sum in the inner loop. The matrix update version is given in Algorithm 3, this implementation both read and writes to matrix A concurrently. If A resides in global memory as it would in a GPU, this can cause problems because the operations have to be executed in the right order. This right-looking column based method is often used within parallel architectures because the inner loops can be fully unrolled. For instance, the j loop multiplies a constant $1/L_{ii}$ with a vector A_{ji} that can be done all in the same cycle. The inner loops are dependent on the enclosing loop, cause it to wait for the inner loop to finish before continuing.

Parallelism can also be achieved by using a blocked algorithm. Instead of computing the full factorization in one go, the problem is divided into block to change vector operations to the corresponding matrix operations. This adds overhead as corrections on the blocks are required, but allows for parallel architectures to divide the work across compute units. There is still sequential behavior in the blocked algorithm but on a inter block scale, not within a block. Switching to a blocked algorithm will change memory access patterns, more read operations are necessary than with a non-blocked algorithm as data is duplicated to multiple compute units. In Figure 4.6 the right-looking column based algorithm with a running sum is visualized, the reads for the sum are light gray, and the output writes are dark blue.

Algorithm 3: Right-looking column based Cholesky factorization

Data: $A \in \mathbb{R}^{n \times n}$
Result: $L \in \mathbb{R}^{n \times n}$
for $i = 0$ **to** $n - 1$ **do**
  $L_{ii} \leftarrow \sqrt{A_{ii}}$;
  for $j = i + 1$ **to** $n - 1$ **do**
    $L_{ji} \leftarrow A_{ji} / L_{ii}$;
    for $k = i + 1$ **to** j **do**
      $A_{jk} \leftarrow A_{jk} - L_{ji} L_{ki}$;
    end
  end
end

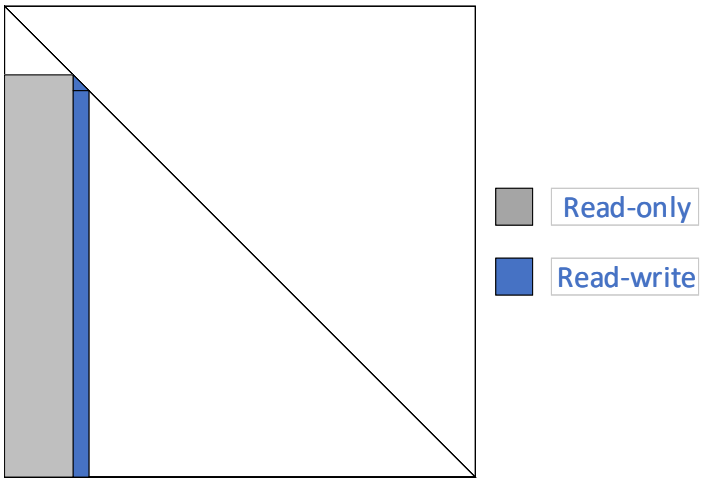

Figure 4.6: Right-looking column based Cholesky factorization

Triangular solver The triangular solver is used while performing a backward or forward substitution where A is the triangular matrix produced by the factorization. A with $Ax = b$ where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. x is defined as a function of A and b:

$$\begin{aligned} x_0 &= b_0/a_{00} \\ x_1 &= (b_1 - a_{10}x_0)/a_{11} \\ x_2 &= (b_2 - a_{20}x_0 - a_{21}x_1)/a_{22} \\ &\vdots \\ x_i &= (b_i - \sum_{k=1}^{i} a_{ik}x_k)/a_{ii} \end{aligned} \tag{4.5}$$

Notice that each new element of vector x depends on all previous elements. This is a data dependency that limits parallel implementation. A simplified implementation of the triangular solver is given in Algorithm 4. Where we have a running sum on b where the correction of previous x_i multiplied with vector A_i is subtracted. This operation can be executed in parallel as there are no dependencies.

A fully unrolled implementation of Algorithm 4 where each element of b has a dedicated accumulator requires a large number of PEs. It performs a MAC operation for each row of A, as the first row only requires a single accumulation subsequent additions are wasted. Depending on the size, the number of PEs required be a large. In our case, $n - 1 \approx 200$ PEs would be required, this is a small number for a GPU but not at all small for FPGA. The minimum number of sequential operations needed for this algorithm is $2n - 1$ because of the dependency between each diagonal element and the previous row. To achieve this we would need the maximum number of PEs (n) which would see an average utilization of 50%.

Algorithm 4: Column parallel forward substitution

Data: $A \in \mathbb{R}^{n \times n}$ where A_0 is column 0 of A, $b \in \mathbb{R}^n$
Result: $x \in \mathbb{R}^n$
for $i = 0$ **to** $n - 1$ **do**
  $x_i = b_i / A_{ii}$;
  $b = b - x_i A_i$;
end

There is little difference between the backward and forward substitution, only the L matrix is mirrored in the diagonal to give U used in the back substitution. If flip the U matrix and input vector we can use the exact same method as in the forward substitution. This allows us to only keep one copy of L in

memory by changing the access pattern, instead of storing both L and U separately. This would only be beneficial if L is kept in local memory instead of global. In most cases the forward substitution is immediately followed by a backward substitution when solving linear systems using factorization. The operations can be made equivalent by flipping the input L instead of using U allowing us to use same PEs. The back substitution has to wait until the forward substitution finishes. As the triangular solvers are sequential and block pipelining as mentioned above, it will have significant impact on the critical path. It is necessary to exploit parallelism where possible, trading in resources for faster execution.

Inter operation sequence Most of the operations in qpAS shown in Algorithm 1 have data dependencies on prior steps. For instance, the forward substitution depends on the Cholesky factorization. In turn, the backward substitution uses the output of the forward substitution as input. The overall operation should be executed in sequence, but input data can become available before the whole operation is done. If compute pipelines are correctly defined then the following operation can already start. For instance, if a column based method as given in Algorithm 3 is used to compute the lower triangular L. Each new column can be passed to a column based triangular solver as given in Algorithm 4. Thus creating a inter-routine pipeline of results, where the output can immediately be used for subsequent operations.

$$\begin{pmatrix} a_{00} & 0 & 0 & 0 & 0 & 0 \\ a_{01} & a_{11} & 0 & 0 & 0 & 0 \\ a_{02} & a_{12} & a_{22} & 0 & 0 & 0 \\ a_{03} & a_{13} & a_{23} & a_{33} & 0 & 0 \\ a_{04} & a_{14} & a_{24} & a_{34} & a_{44} & 0 \\ a_{05} & a_{15} & a_{25} & a_{35} & a_{45} & a_{55} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}$$

Figure 4.7: Triangular linear equation solver $Ax = b$, where A and b are known.

4.3.3. Variable iterations of control loop

As discussed in previous sections, the optimization problem checks at the end of each iteration if the current solution satisfies the requirements. If all requirements are satisfied, the solution is returned. If the solution does not satisfy the requirements and the maximum number of iterations is not yet reached a new iteration will start. The maximum number of iterations exists to limit the execution time of the algorithm.

Having a variable bound on loops complicates the unrolling of loops and pipelining. Not knowing how many times a loop can be unrolled limits parallel implementation and introduces control logic making loops sequential. Because our Algorithm uses an Active-Set method, these variable bounds are present in almost every step. In Algorithm 3, the loop bounds depend on other loop variables that are dependent on the matrix size n. This size is not constant, and changes based on the size of the Active-Set. It is possible to get predictable bounds by always using the maximum, but this depends on the implementation if it will be beneficial. For an FPGA this would create more PE which are scarce, making this expensive option.

4.3.4. Data representation

The data representation used is in the algorithm is double precision floating-point. The reason for this is that most work on optimization algorithms is done in MATLAB, which uses double precision by default. A good practice when using FPGA accelerators is changing the data type to a fixed-point representation or single precision floating-point. This is largely because there is still no hardened support for double precision operations in modern FPGAs architectures in contrast to single precision and implementing this in logic is costly in resources. This means that if FPGAs are considered, double precision implementations get less relative performance compared to a GPU or CPU.

The accuracy of the input and output of the algorithm is not the only parameter involved in switching from double to single precision. For the Cholesky factorization there is a risk that a matrix becomes singular when moving from double to single precision [8]. In our case the input Hessian matrix H becomes singular in single precision therefore the factorization of this H gives a wrong result. If we take the double precision factorization L of H, it can be converted to single precision because the operation is not part of the critical path. This single precision version of L lest the algorithm produce the same results while lowering the precision. Each change of H and input x influences the singularity, as such it does not work for all cases, which is why double precision is commonly used. An alternative to floating-point, especially for FPGA is moving to a fixed-point representation that meets the requirements given by Demmel [8]. This would greatly improve the possibilities for FPGAs as the floating-point throughput is bounded by the number of DSP available. However, the maximum throughput in FLOPS/s of modern FPGAs is still low compared to GPUs.

It is possible to use a mixed-precision implementation of the algorithm where only the necessary parts like the factorization use double precision, and the other operations with less requirements use single precision. However, the impact on the algorithm accuracy of such optimizations is unclear and therefore out of scope for this thesis.

4.4. Alternative solutions

This section aims to determine what hardware platform is best suited to offload parts or the complete qpAS algorithm. From Chapter 1 we know the platform chosen for the implementation is already set by the scope of the thesis. It is still important to highlight how the algorithm would map to different platforms. First, the CPU is evaluated for its flexibility and no offloading penalty. Second, the GPU is considered with its high number of cores and floating-point capabilities. Last, the FPGA is chosen because of its dataflow approach.

4.4.1. CPU

Currently, the CPU is the default deployment platform because of the effort required to rewrite algorithms to different languages and platforms. If the current CPU is not capable of meeting the requirements, the algorithm could still be offloaded to a dedicated CPU that offers more cores or a higher clock frequency. This section will cover the aspect of the qpAS algorithm that are well suited for CPU execution compared to the other platforms.

There are several properties of the algorithm that would suit the CPU over other options. Sequential execution is something that comes naturally to the CPU architecture, with its high clock frequency and dedicated floating-point arithmetic units to deliver maximum single-thread performance. Because of the sequential behavior in the algorithm this clock frequency plays a significant role in overall performance. A CPU is clocked 5-10 times higher than most FPGA designs and 2-4 times compared to most GPUs. This frequency does come at the cost of having a lower number of cores available, this is why GPUs are more suited for massively parallel problems. Our use-case is hybrid in that some parts can be done in parallel and others still need to wait for previous results. This causes a trade-off between allowing the algorithm to run in parallel and lowering the execution time of a single computation.

If we loop at the algorithm optimizations that are currently not used in the CPU implementation most try to reduce the number of compute operations required by introducing a more difficult control flow. Reducing the factorization to only add a single row requires reading and writing to existing memory structures with non-regular patterns which the CPU can do without effort.

4.4.2. GPU

Offloading to a separate board to take advantage of the characteristics makes sense when the computation time is long enough to notice the benefits. In our use-case the deadlines are short (in the order of 20-40 ms). Interconnect bandwidth requirements are low because each run of the algorithm reuses parts of the data. If only the mathematical operations are changed from a CPU call to a GPU library function. The required data transfers happen in the critical path as the subsequent operations are not independent. It is possible to move the complete algorithm to a GPU, but as there is control logic and sequential operations, the occupancy of the high number of cores would be low.

4.4.3. FPGA

An increasingly popular solution for offloading the processing of sensor data in industrial applications is FPGAs. The latency of each compute part of the algorithm is known which makes execution time, apart from memory accesses and PCIe transfers, deterministic. This is useful for our use-case as the deadlines are short and cannot be exceeded. Where a GPU has lower efficiency because data parallelism is not present, the FPGA can create a deep pipeline from the sequential algorithm to achieve parallelism and get a higher throughput.

As noted before, the data-type used in an FPGA is of great importance. Fixed-point representations are mapped the best, with single precision floating-point doing well if there is hardened support. In our case the algorithm uses double precision. If it cannot be changed to use fixed-point or single precision the FPGA will have a hard time competing with other solutions that do have double precision support. There are intermediates possible. For instance, the input could still be double precision but the kernels convert the data to fixed or single precision. This requires in depth knowledge of the input data and clear requirements of the precision on output data. Even then, singularities because of the precision cannot be completely avoided.

When moving data from host to device, there is always a penalty. This would have to be small enough to allow for speedup of the algorithm, and not just slowing it down. The main problem is the control decisions, if only a small part of the algorithm is offloaded. The FPGA would have to wait on the decision of the host, before it could execute the next iteration. Ideally there should be speculative execution of the next iteration but with this type of algorithm that is not trivial. Fortunately, the data movement needed between the host and device across iterations is not large. Low memory bandwidth of an FPGA is thus not a problem. However, latency is an important factor to take into account.

Each iteration does perform the same steps in sequence. Once the kernels support efficient methods of dealing with the growth in the Active-Set it becomes important to manage the dataflow between the kernels. Because of the channels feature added by Intel which provides high throughput low latency interconnect between kernels. It becomes simple to have wide and deep channels between the kernels to prevent stalls in the production and consumption of data. This data streaming is an important feature in the OpenCL framework as it improves throughput and removes the need for explicit control.

4.5. Chosen solution

The scope of this thesis was part of the assignment received from ASML, in which the use of FPGA accelerators using a high-level abstraction was specified. There is potential in using an FPGA based accelerator if an optimized implementation can be achieved and data types are adjusted. An optimized FPGA can definitely not compete with a equally optimized CPU implementation if double precision floating-point is used.

The FPGA architecture offers benefits when pipeline parallelism can exploited and data is streamed between kernels to ensure the occupancy in each kernel is optimized. Another benefit is the size of local memory in combination with little changes of the input data between iterations and even across calls. This means data can be kept in local memory reducing the communication requirements between kernels and off-chip RAM. The alternative to an FPGA solution would be improving the CPU implementation to take advantage of more cores and to apply algorithm specific optimizations.

From Chapter 1 we note the following requirements on the FPGA solution:

- High-level abstraction such as OpenCL
- Leverage the features of FPGA without intricate knowledge about the underlying hardware

It should use a high abstraction level, something an algorithm developer without knowledge about FPGAs could use. We evaluate the performance achieved of using algorithm and architecture based optimizations.

There are two options for offloading the algorithm: implement the complete algorithm or only the intensive parts. First we start with implementing the hotspots, if the complete algorithm is offloaded these are necessary anyway. The hotspots that we start with are: refactorization of the lower triangular matrix (cholapp) and the triangular solvers. Cholapp because it is identified as the largest growing factor across iterations, and the triangular solver because it is a significant portion of the static execution time present in each iteration. The next extension would be the matrix-vector multiplication Adx, this runs well on CPU and GPU but may impact the FPGA more. This depends on the size of both the A matrix

and dx, if they are large enough, the FPGA needs to use a significant portion of the resources. This is in contrast with both the CPU and GPU that either have a large number of cores or vector instructions (AVX) available.

The goal is to achieve a speedup for the hotspots to get a faster algorithm execution time. The power consumption and resource usage can be sacrificed to achieve a faster design. In Chapter 5 the implementation of the solution is presented and evaluated.

5 Implementation

In this chapter we introduce our solution to offload the qpAS algorithm to FPGA using the OpenCL framework. We divide the solution into five parts:

1. Memory Interface
2. Cholesky factorization
3. Substitution
4. Matrix-vector multiplication of Adx
5. Control logic

The implementation focuses on use-case (a) from Section 4.2 as a starting point. In our implementation we only consider adding constraints for the test as this causes the highest execution time. Efficient downdate and correction methods can be used to reduce the work required for removing a constraint. Additionally, The Adx matrix-vector multiplication is not required when removing a constraint.

First, we start with an overview of how the parts mentioned above fit together and what data from global memory is needed. Second, we elaborate on the flow of the algorithm and data within the FPGA. Last, we look at the computational kernels that are fed the data and compute the results.

5.1. Overview

In this section we provide an overview of how the compute kernels fit together and what data communication is required between them. This is important to understand how parts fit together and how to balance them in terms of resources and latency. A block diagram of the system is shown in Figure 5.1. Some algorithm based optimizations are already applied to this implementation. Determining the factorization of Hessian H and A_{as} is already split to reduce the work of the factorization. Additionally, L is precomputed using the upper Cholesky factorization of the Hessian H. B is precomputed by solving the linear system of equations $L \backslash B = A$. Using B and L a more efficient refactorization is done by only adding or removing one line each iteration. Another algorithm optimization used in our implementation is in the forward substitution. The first n values of the vector y are precomputed and present in global memory. The forward substitution is only computed for each new line of L. The backward substitution completely changes upon each iteration and will be fully computed.

5.1.1. Autorun and enqueued kernels

There are two types of kernels in the system, enqueued kernels which are queued from the host side and autorun kernels which start to run as soon as the device is configured. In Figure 5.1 we see that only the read kernels and the controller are enqueued, the others are autorun kernels. As soon as an autorun kernel finishes executing, it will start from the beginning. These kernels wait until data is present in their channels to immediately start processing when it arrives. As soon as the kernel produces a result, it is pushed to the output channel. The next autorun kernel waiting on the input can then read from that channel to create a stream of data. When the result of the back substitution is done it is passed to the controller, which initiates the control logic and starts a new iteration.

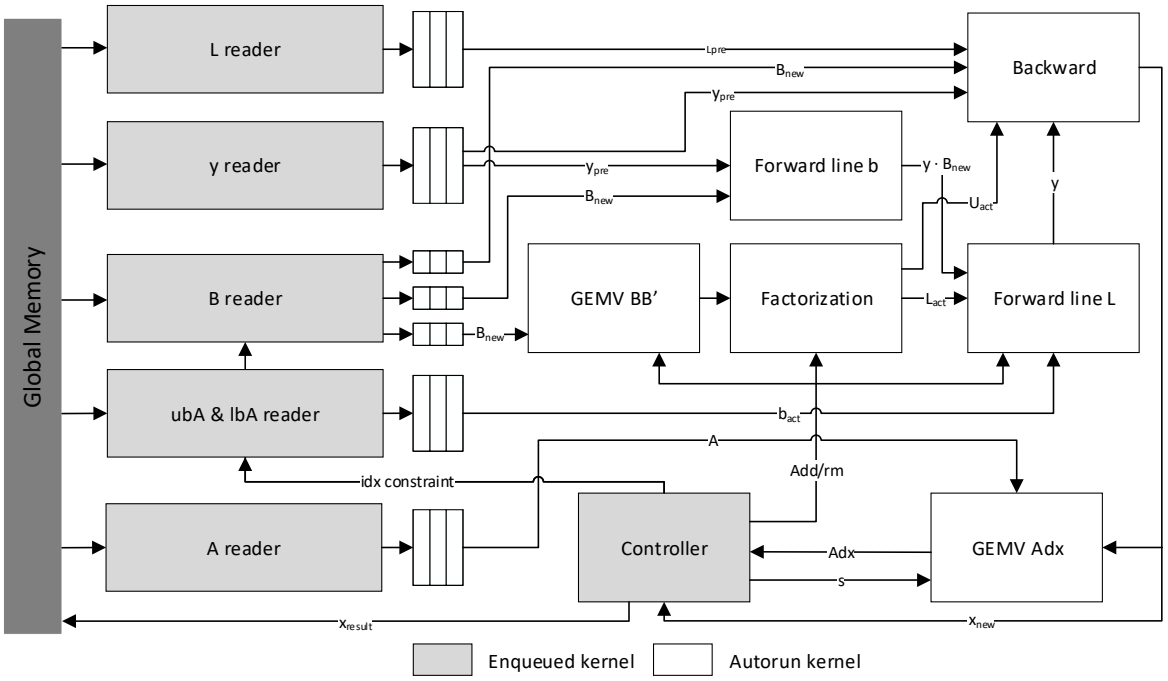

Figure 5.1: Diagram of the qpAS implementation using OpenCL kernels

5.1.2. Channels

Channels provide direct communication between kernels, and are the only method autorun kernels have to communicate. There are two different types of channel operations: blocking or non-blocking. The difference is that for a blocking read or write the pipeline will stall until the operation is complete, while the non-blocking has a return value that indicates whether the operation was successful or not. To ensure data is sent in the correct order, it makes sense to always choose a blocking version. This way an autorun kernel will block until data is received. We can define the depth of channels, this allows the user to push data into the channel without it blocking if the data is not read immediately. If a channel is full, the kernel that is writing to the channel will block until space is available. This is not a problem for the kernels that read from memory, as their only job is pushing data into the channel, if they are blocked no functionality is lost.

In our implementation we use explicit synchronization to control the autorun kernels. The kernel reads and writes are designed in such a way that the operations always happen in the correct order. We ensure that all required input data is first read, only then will the autorun kernel perform its operations. This design in combination with deep channels ensure that pipelines are not blocked.

Kernel pipelines might stall when a compute kernel, such as the factorization kernel produces data faster than the kernel that reads from the channel consumes it. Making sure that pipelines are not blocked requires fine tuning the number of PEs of each kernel. It is important to create a balance between consuming and producing. To mitigate some of the impact of unbalanced kernels deeper channel can be used to buffer the data.

There is an option in the SDK from Intel to have vectorized channels, in which independent channels are created that each can be written and read at the same time without blocking the others. This is especially useful when unrolling loops in which there is a channel read, each CU can get their own channel to allow for concurrent reads. A risk of using blocking channels is that the kernels can deadlock if a read-write loop is created by the compiler. In this situation a kernel is blocked writing to a channel, while the subsequent kernel is blocked on a different channel. This is an unrecoverable situation, requiring the user to reconfigure the FPGA. Our implementation makes sure pipelines are not blocked as described above.

5.2. Memory interface

An important aspect of the implementation is handling memory, where data is stored and when data is retrieved from global memory. There are matrices that only have to be loaded once per run of the algorithm. These are the static matrix L of Hessian H and the precomputed result of the forward substitution y_pre. For the matrix B used in $B * B^{\mathsf{T}}$ only the vectors in the Active-Set B_w are needed. As the active set can increase or decrease each time a new vector is retrieved from global memory it is pushed to the channels connected to multiple kernels. The indexes of the Active-Set are passed by the controller to the read kernels using similar channels which have a depth of 1. The read kernel of the A matrix has to provide new data for the Adx matrix-vector multiplication kernel which receives dx in a streaming fashion. If A is of size $m \times n$, each new element $dx[i]$ needs m values of corresponding with index i = $A[0..m-1][i]$. The output Adx is only valid after all n values of dx are processed, stopping us from further streaming the solution to the controller.

To decrease the continuous load on the global memory for kernels that read in a repeated pattern a cache is created by the SDK. This only happens for kernels where the data does not change in global memory while it is executing. This decreases access time for pushing data in the channels and allows for prefetching as the access pattern is known upfront.

After each iteration the controller checks x_{new} to see if the found solution satisfies the requirements. If this is the case, x_{new} is written to global memory and the solution can be transferred back to the host.

5.3. Compute kernels

In the following five sections the main computational kernels are presented. Not all the blocks as shown in Figure 5.1 are separately mentioned. However, all kernels are included in the section of the computation they contribute to. Each kernel is different from the CPU model, some kernels are made to allow streaming data while others use parallel execution. The memory and kernel communication interfaces, which are mainly channel reads and writes, are explicitly highlighted.

5.3.1. Cholesky factorization

This kernel replaces the cholappend functionality from the CPU implementation used in Section 4.2. The BB^{T} and line factorization kernel as shown in Figure 5.1 together perform this operation. The old cholappend function had two purposes: to determine the input of the Cholesky factorization by multiplying $B_{as}B_{as}^{\mathsf{T}}$ and computes the lower triangular matrix of that result using the Cholesky factorization.

Algorithm 5: Matrix-vector multiplication BB^{T} kernel implementation high-level

Input: B_{new}
Data: $B_{as} \in \mathbb{R}^{MAX\_ITER \times MAX\_ITER}$
Result: $(BB^{\mathsf{T}})_{new}$

1 n_as = read_channel$_{n_as}$;
2 $\vec{acc} = 0$;
3 **for** $i = 0$ **to** n **do**
4 $\quad B_{new}$ =read_channel$_{B_{new}}$;
5 $\quad \text{col}_i(B_{as}) = \begin{pmatrix} \text{col}_i(B_{as}) \\ B_{new} \end{pmatrix}$;
6 $\quad \vec{acc}$ += $\text{col}_i(B_{as}) \cdot B_{new}$;
7 **end**
8 write_channel$_{BB^{\mathsf{T}}}$($\vec{acc}$);

The pseudocode of an OpenCL implementation of updating the matrix multiplication BB^{T} is presented in Algorithm 5. As every iteration only a single vector is added to B_{as} compared to redoing a complete matrix-matrix multiplication, this operation can be substituted by adding the result of a matrix-vector multiplication as a new row to the previous result. When a new row is added, the dot product of the new vector with itself and all others in the set is computed. Each new row thus means one more dot product is required, resulting in a triangular shaped matrix. Traditionally a matrix multiplication produces a square matrix. Using only the triangular matrix still works because the result of BB^{T} is

symmetric by definition. In this case the lower triangular contains all necessary information. Similarly to most other autorun kernels execution is blocked until data is present in the channel that provides information about the size of the Active-Set n_as. This information is used to add constraints to the set, provide information to the read kernels to fetch the new vector and push that data to the compute kernel. On line 4 the new vector of B_{as} is read from the channel and written to a local copy. The new row of BB^{T} is calculated by performing a dot product between the new vector with all other vectors in the set including itself. As these products are independent, multiple CUs can be added to compute the products in parallel. With the OpenCL unroll pragma the acc vector operation can easily be fully unrolled. CUs are added each position in the acc vector, for all accumulators not used by the Active-Set empty products are produced and thus ignored. Unrolling here removes the loop normally introduced when doing vector operations with a single CU. With the unrolling, no nested loops are present and the II is reduced to 1. Each cycle a value of B_{new} is read and fed into the compute pipeline.

Algorithm 6: Factorize line

Input: B_{new}
Data: $L \in \mathbb{R}^{MAX\_ITER \times MAX\_ITER}$
Result: $(BB^{\mathsf{T}})_{new}$

```
1  n_as = read_channel_{n_as};
2  sum = 0;
3  for i = 0 to n_as − 1 do
4      BB^T = read_channel_{BB^T};
5      acc = 0;
6      for j = 0 to i do
7          acc −= L_{n_as,j} L_{ij};
8      end
9      L_{n_as,i} = (BB^T + acc)/L_{ii};
10     sum += L²_{n_as,i};
11     write_channel_{L_as}(L_{n_as,i});
12 end
13 BB^T = read_channel_{BB^T};
14 L_{n_as,n_as} = sqrt(BB^T − sum);
15 write_channel_{L_as}(L_{n_as});
```

The OpenCL kernel that adds a row to the lower triangular is functionally described in Algorithm 6. There are two main loops in this kernel, each with an accumulator. It is the sum of all squares of L except for the last value, which is the diagonal. The accumulator acc is used to compute all non-diagonal values of L, this loop runs from $j = 0$ to the current column $j = i$. This complicates adding more compute units as the loop bounds are not consistent. Unrolling to the maximum like in the BB^{T} kernel is not a possibility, the data outside of the loop is not always empty and thus will have impact. Instead, it is important to ensure the II is one to still get good performance. The inner loop only contains a MAC operation, if the compiler recognizes that before the loop the accumulator is set to zero, the compiler will implement an efficient MAC unit which has an II of one. The for loop over a column cannot be pipelined, as there is a data dependency between computing a value on line 9 and reading it again on line 7. Each time a new value is computed, it is directly pushed to the output channel to subsequent autorun kernels.

5.3.2. Forward substitution

The forward substitution also consists of two kernels: the ”Forward line b” kernel computes the dot product of the static solution y_pre and B_{new} while ”Forward line L” determines a solution based on the dot product and Active-Set factorization. These kernels are split to keep the pipelines simple, when a new index of the Active-Set is known, the B reader kernels will push that data into the ”forward line b” kernel which computes the dot product and passes the result to the ”Forward line L” kernel. Although the kernels could be combined, we have chosen to create two separate kernels to ease unrolling and pipelining.

Algorithm 7: Dot product of y_pre and B_{new} kernel functionality

Input: y_pre, B_{new}
Data: $y\_pre \in \mathbb{R}^n$
Result: $dot(yb)$

```
for i = 0 to n do
    y_pre = read_channel_{y_pre};
end
while iter < MAX_ITER do
    acc = 0;
    for i = 0 to n do
        acc += y_pre_i · read_channel_{B_new};
    end
    write_channel_yb_sum(acc);
    iter++;
end
```

The autorun kernel described in Algorithm 7 does not actually run multiple times. Instead, it has an internal while loop that simulates the same behavior. This choice was made in order to allow the kernel to only read the data in y_pre once over multiple iterations. Another option would be using a token-based system as seen for x0 in Algorithm 10. This would create more control signals from the controller, as this information is in addition to the currently used signals. With this implementation, when data is presented on the y_pre channel it can be read into local memory, even before the controller itself is enqueued. The kernel functionality is very basic, it only computes the dot product between the static vector y_pre and dynamic B_{new} which arrives in a streaming manner from the B reader kernel. After all n elements of B_{new} are processed it will push the data to the output channel.

Algorithm 8: Forward substitution of Active-Set part kernel functionality

Input: bound, yb_sum, n_as
Data: sol $\in \mathbb{R}^{MAX\_ITER}$
Result: y_as

```
n_as = read_channel_{n_as};
bound = read_channel_bound;
yb_sum = read_channel_yb_sum;
acc = 0;
for i = 0 to n_as − 1 do
    acc += sol_i · read_channel_{L_as};
end
sol_{n_as} = −(bound + acc + yb_sum)/read_channel_{L_as};
write_channel_{y_as}(sol_{n_as});
```

The part of the forward substitution that uses the lower triangular of the Active-Set is explained in Algorithm 8. This kernel has multiple input channels that provide the necessary data. Like other kernels, n_as is read first as this is the kickoff signal from the controller. Afterwards the new part of the b vector is read from the bound channel, which contains a value from either ubA or lbA according to the new index of the Active-Set. The partial sum output of Algorithm 7 is added to the dot product of the Active-Set L and previously computed solutions. One new solution is calculated each time the kernel runs, alternatively a solution can be removed and subsequent values can be updated.

5.3.3. Backward substitution

The backward substitution kernel given in Algorithm 9 breaks the streaming behavior flow from the previous kernels. It first has to aggregate all input data before it can start. Large local memory buffers are present to store L and B_{as} which are relatively large. The number of block RAMs used to save these data sets is relevant when unrolling compute loops as more data is read each cycle. If more read ports

are required to feed data into the PEs, partitioning of data is necessary. An option is to use double pointers, sized to the next power of two, which allows us to select clear bits on which the memory should be partitioned. For example, if the unroll factor is 32 and the data structure is $L \in \mathrm{R}^{256\times256}$, then 32 concurrent accesses are required on the last 5 address bits of the second pointer. With an attribute the compiler can be instructed to bank the memory structure on those bits, which allows us to read the correct 32 addresses from separate blocks in one cycle.

The data that remains static across iterations is read from channels before entering the while loop as to not repeat this each iteration. As the Active-Set grows, at the beginning of an iteration a new vector is added to the active set, accompanied by a new row in the factorization and a new solution from the forward substitution. From the methods discussed in Section 4.3.2 the column based method is chosen to ease exploiting parallelism. At the beginning of each column a new solution is computed and then multiplied with all other values in that column and added to their respective partial sums.

Adding more CUs to partial sums is a trade-off between resource usage and gain in throughput. Using more DSPs is not a limiting factor, because at most only 20% are utilized. However, creating enough read ports on local memory structures B and L to feed the DSPs with the required data is limiting unrolling further. Each RAM block can have several read ports, overall if all three for loops that compute partial sums are fully enrolled around 300 RAM blocks are needed to provide the read ports required. The resource impact is not the only aspect affected, additional the increased routing also influences the clock frequency. This lowers the performance of the critical sequential parts in other kernels, as all kernels are in the same clock domain. Adding CUs using pragmas requires powers of two but in the case of fully enrolled loops it can deviate from those powers. The OpenCL compiler tries to pipeline as much as possible, so even if loops are executed sequentially they do not re-use CUs from previous loops. It is thus not possible to use re-use parts from the loop on line 18 in the loop on line 28.

When a new element of the solution is calculated it is directly pushed to the output channel that is connected to the controller and Adx matrix-vector multiplier. This creates a situation where it is important to create a balanced implementation. If data is produced by the backward substitution faster than Adx consumes it, the additional resources are wasted as they have little to no impact on the execution time.

5.3.4. Matrix-vector multiplication Adx

The last compute kernel is the Adx matrix-vector multiplication. Contrary to the other kernels no information about the Active-Set is needed. It computes the product of matrix $A \in \mathbb{R}^{m\times n}$ and $dx \in \mathbb{R}^n$, where dx is the difference between x_old and x_new. x_new is streamed from the backward substitution, the moment a new value of x_new is present in the channel the partial sum is updated for all m output values. With $m = 2940$, it is too large to fully unroll because the current target device only contains 1518 DSPs. However, partially unrolling the computation with powers of 2 is possible. The unrolling should increase the throughput to match rate of the input from the back substitution. As the number of cycles between new input data determined by the back substitution, the latency of the loop containing the channel read of dx should complete in the same time as the backward substitution.

In this kernel the matrix A is not stored in local memory, since in some test-cases storing A would require around 50% of the available BRAMs. Instead, we chose to have a read kernel that fills deep channels with the required data from A such that a channel read will never stall the pipeline. When more CUs are added more channels are also added to provide the parallel reads. This shifts the problem of RAM usage from the compute kernel to the read kernel. Because the compiler notices a regular access pattern as the A matrix is always read in the same manner, it performs burst reads from global memory to build a local cache. Doing this prevents the compute kernel from running stalling on the channel reads from A. Another benefit of this split is that the buffers can already be filled with data before the backward substitution is done. As the depth is chosen to the maximum number that corresponds to the size of a single RAM block, which can store 20 kilo bits.

5.3.5. Control computations

Next to the main compute kernels there are some other computations present in the algorithm, these are mostly BLAS level-1 operations such as multiplying a vector with a scalar or checking for a minimum or a maximum in a vector. These operations require relatively little compute, and can be processed in a pipelined as the data arrives from both the back substitution and the matrix-vector multiplication.

Algorithm 9: Backward substitution kernel functionality

Input: $y\_as, y\_pre, B\_new, L, L\_as, n\_as$
Data: $y\_pre \in \mathbb{R}^n$, $y\_as \in \mathbb{R}^{MAX\_ITER}$, $L_{as} \in \mathbb{R}^{MAX\_ITER \times MAX\_ITER}$
Data: $L \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{MAX\_ITER \times n}$
Result: x

$n\_as =$ read_channel$_{n_as}$;
for $i = 0$ **to** n **do**
  for $j = 0$ **to** n **do**
    $L_{ij} =$ read_channel$_L$;
  end
  $y\_pre_i =$ read_channel$_{y_pre}$;
end
while *iter* < *MAX_ITER* **do**
  partial$_as = 0$;
  partial $= 0$;
  for $i = 0$ **to** n **do**
    $B_{n\_as,i} =$ read_channel$_{B_new}$;
  end
  for $i = 0$ **to** n_as **do**
    $L_{n\_as,i} =$ read_channel$_{L_{as}}$;
  end
  $y\_as_{n\_as} =$ read_channel$_{y_as}$;
  for $i = 0$ **to** n_as **do**
    sol $= (y\_as_i +$ partial$_as_i)/L_as_{ii}$;
    write_channel$_x$(sol);
    for $j = 0$ **to** n_as **do**
      partial$_as_j$ += L_as_{ij}sol;
    end
    for $j = 0$ **to** n **do**
      partial$_j$ += B_{ij}sol;
    end
  end
  for $i = 0$ **to** n **do**
    sol $= (y\_pre_i -$ partial$_i)/L_{ii}$;
    write_channel$_x$(sol);
    for $j = i + 1$ **to** n **do**
      partial$_j$ += L_{ji}sol;
    end
  end
end

Algorithm 10: Matrix-vector multiplication Adx kernel functionality

Input: s, x, A
Data: partial $\in \mathbb{R}^m$, $x\_old \in \mathbb{R}^n$
Result: Adx

```
1  partial = 0;
2  if read_channel_{init_x_old} then
3  |  x_old = 0
4  end
5  s = read_channel_s;
6  for i = 0 to n do
7  |  dx = read_channel_x − x_old_i;
8  |  x_old_i += s · dx;
9  |  for j = 0 to m do
10 |  |  partial_j += dx· read_channel_A;
11 |  end
12 end
13 write_channel_Adx(partial);
```

This pipelining can hide the effect on the overall execution time. The control operations required are described in Algorithm 1 from line 14-21 and 24-29. When a new index is determined it is sent to the B reader and the compute kernels receive the new value of n_as as a start signal for the new iteration.

5.4. Host code

The host machine manages the FPGA by configuring the kernels using PR and transfers the input data to the FPGA and reads back the result. The host will transfer newly measured input data to the global memory on the FPGA board, and then enqueues the controller to start the algorithm. Our use-case studies the execution time of a single run of the algorithm, thus simplifying the host code as the kernels are only enqueued once. There are some remarks when reading or writing from the host to the FPGA. The PCIe card allows users to make **Direct Memory Access** (DMA) reads and writes, reducing the time necessary. A precondition for DMA operations is a 64 byte alignment on the host side memory structures. All the read kernels are executed in separate command queues for concurrent execution. The controller is enqueued last, as to ensure that all read kernels are ready to read from their channels. Once the controller reaches the maximum allowed iterations or finishes executing, the host copies the data back from the board with a DMA read. The time it takes for kernels to start and transfer data should be included in the accelerator use-case to make a fair comparison between the deployment strategies.

6

Results and evaluation

In this chapter we evaluate our implementation and compare it to the reference CPU design and to GPU library calls. First, we present the test environment, the data used, the test setups and evaluate the impact of input data. Second, we present measurements done on the FPGA OpenCL implementation. These results are an in depth analysis of optimization techniques without a comparison to alternative hardware platforms. Third, we compare the performance of the CPU to the GPU and the FPGA. Last, we evaluate the achieved performance, identify possible improvements of the FPGA implementation and evaluate if some optimizations done can also be applied the other hardware platforms.

6.1. Experimental setup

In total there are three experimental setups used to represent the different hardware platforms. The CPU system used for profiling the reference implementation is described in Table 4.1. The system that has an FPGA accelerator PCIe board is given in Table 6.1. Lastly, the system equipped with a GPU is specified in Table 6.2.

Table 6.1: The setup used for OpenCL FPGA tests

Item	Value
CPU	Intel Xeon E5-1620v2 (4 cores 8 threads @ 3.7 GHz)
RAM	32 GiB DDR3 1866 ECC
OS	RHEL 7 (64-bit)
Compiler	GCC 4.8.5
FPGA SDK	Intel FPGA SDK for OpenCL 19.1
FPGA	Intel Arria10 GX 1150 devkit
Interconnect	PCIe 3.0 8x (8 GB/s)

Table 6.2: The setup used for GPU tests

Item	Value
CPU	Intel(R) Xeon(R) Gold 6134 (8 cores 16 threads @ 3.2 GHz)
RAM	192 GiB DDR4 2666 ECC
OS	RHEL 7 (64-bit)
Compiler	GCC 4.8.5
CUDA	10.1
GPU	NVIDIA(R) V100 SXM2
Interconnect	PCIe 3.0 16x (16 GB/s)

The input data of the test-case is the same as use-case (a) in Section 4.2. The sizes of the input data and whether the data remains static or changes across iterations are given in Table 6.3. This

use-case has a relatively low number of iterations but a large Adx matrix-vector multiplication. In order to compare the FPGA and GPU implementations to the CPU implementation, data transfers to and from the FPGA or GPU are presented separately. Execution time is measured on the host-side, CPU timers are used to measure the GPU and FPGA function calls. All execution time measurements are averaged over 15 runs to reduce the effect of outliers. In the CPU case MKL functions are called for the BB^{T}, Cholesky, Substitution and Adx. These functions are also present in the cuBLAS library provided by NVIDIA for their GPUs. The OpenCL FPGA implementation is different, as it does not use library functions, but instead uses our custom implementation. Most of the optimizations present in the FPGA implementation are not portable to library calls and are therefore not present in the CPU and GPU implementations. These optimizations can be implemented in custom calls or code for the other platforms, which could improve their performance.

Table 6.3: Input data specifications for use-case (a)

Input data	Size	static/dynamic
A	2940×215	static
B	2940×215	static
ubA	2940	dynamic
lbA	2940	dynamic
L	215×215	static
x0	215	dynamic
y_pre	215	dynamic

All the presented FPGA results are using single precision floating-point data types unless specified differently. This choice was made to take advantage of the hardened floating-point support for singles present in the FPGA architecture. The CPU and GPU results use double precision floating-point data types as this is default in the reference implementation. The FPGA produces the same output in double and single precision. This raises the question if double precision is actually necessary for the dynamic parts of the algorithm. As mentioned in Chapter 4, the Cholesky factorization of H does not work in single precision for data sets that become singular after conversion [8]. If the precomputed factorization of H is converted to single precision, it can be used to get a correct result from the algorithm for our use-case.

6.2. FPGA results

In this section results of the OpenCL FPGA implementation are presented. We apply multiple optimizations. We start with accelerating the main bottleneck after which we move on to smaller optimizations.

As mentioned in the previous section, the matrix-vector multiplication of A and dx is significantly large in our use-case. In Figure 6.1 the execution time is shown for all cases where none or only one of the kernels includes unrolling. The execution time decreases by a factor 10 when CUs are added to Adx. With only one CU for Adx the kernel cannot process the incoming data fast enough and therefore it dominates the execution time. With 32 CUs added there is already a 10x speedup of the overall execution time. To see the impact of adding CUs to other kernels, measurements where only one CU is given to Adx are ignored.

6.2.1. Effect of unrolling loops

In the OpenCL for FPGA framework, loops can be unrolled to add compute units at the cost of additional hardware. In the qpAS algorithm there are three kernels that are candidates for unrolling as they have independent operations suitable for parallel execution. Usually, fully unrolling loops has limited benefit as the compiler can generate significantly more hardware to gain a single cycle. The other option is to partially unroll loops to find a trade-off between resources and performance. Partial unroll factors have to be powers of two in the Intel OpenCL SDK. Four different unroll factors are used for Adx: 16, 32, 64 and 128. Larger would not make sense as n is smaller than 256. Unroll factors lower than 16 are not included as these would only give marginal benefit while introducing unrolling overhead. Most loops that can be unrolled have n as their upper bound. In our use-case $n = 215$, so loops unrolled with factor 32 require 7 iterations as $7 \times 32 = 224 \geq 215$ is sufficient. Unrolling can cause the latency or II of a loop

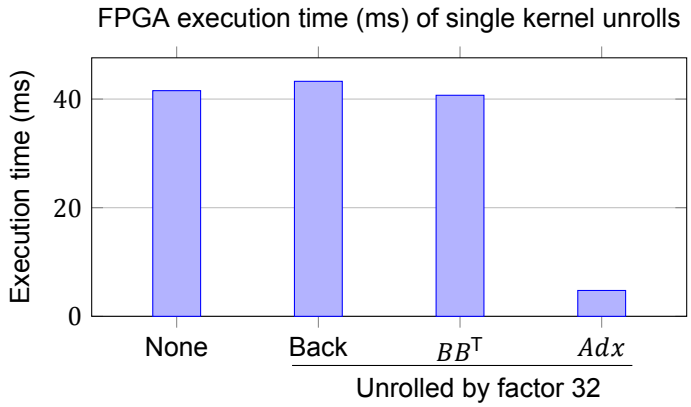

Figure 6.1: Impact of unrolling single computational kernels for use-case(a)

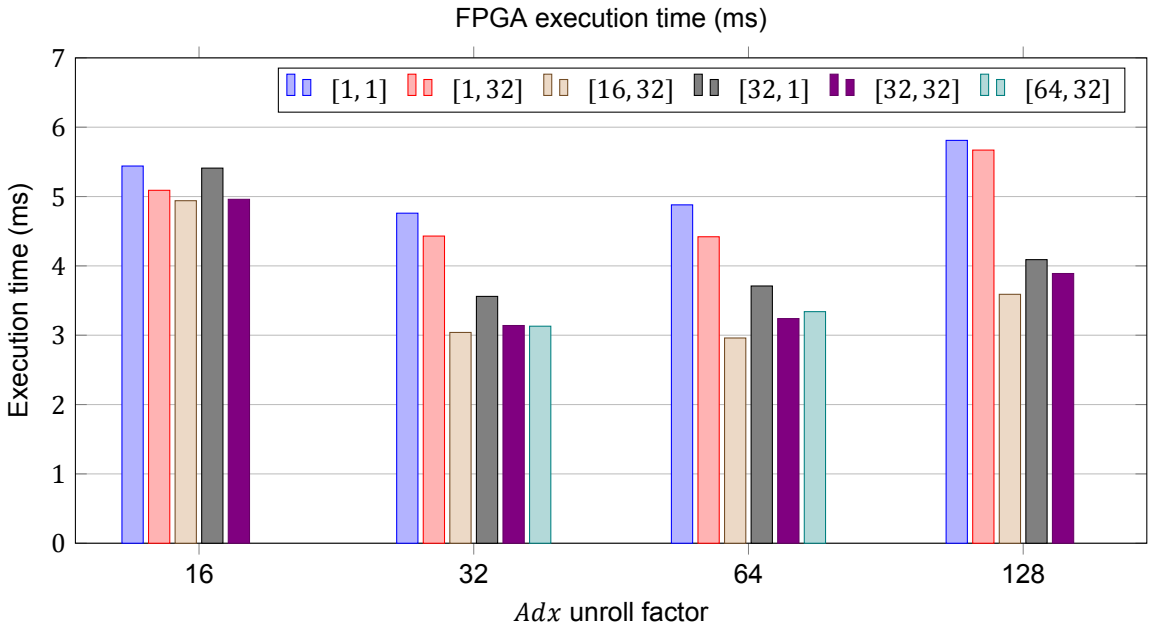

Figure 6.2: Design space exploration of the execution time against unrolling combinations sorted on Adx unroll factor for use-case (a), legend: [unroll back sub, unroll BB^{T}]

to change, this depends on the operations present in the loop. The benefit of unrolling in those cases can be less than expected, because adding twice the number of CUs can bring less than two times the throughput. In this case more effort is required to change the structure of the operations to avoid dependencies and lower the II to one for the inner loops. In Figure 6.2 the execution time for different unroll factors of Adx are shown, the colors of the bars represent different unrolling combinations of the back substitution and the BB^{T} matrix-vector multiplication.

Once the main bottleneck Adx matrix-vector multiplication is solved, the balance of kernels comes into play. For 16 Adx CUs unrolling BB^{T} improves performance while unrolling the back substitution has almost no impact. In this case, backward substitution kernel is still producing faster than Adx consumes the data. Evidence of this is found when looking at the results of unrolling Adx by factor 32: when CUs are added to the back substitution, Adx consumes data fast enough to drop the overall execution time. There is limiting factor that prevents the algorithm from going any faster even if more CUs are added to the kernels currently being unrolled. This behavior is explored in Table 6.4, where multiple baseline unroll configurations are evaluated in execution time, frequency and resources. In this table it can be seen that adding more CUs to the back substitution beyond 16 does not benefit the execution time in any of the configurations. This difference is caused by a lower operating frequency for a high unroll factor of the back substitution. Currently, the backward substitution kernel stalls after reading the static

data, until all the input data is received. To further increase the speed of the implementation, the kernels that feed the back substitution should produce their results earlier. Either the factorization or the forward substitution is currently stalling the pipeline, stopping the algorithm from speeding up further.

Other interesting behavior seen in Figure 6.2 is the increase in execution time when unrolling Adx by 128. If we look at the entry in Table 6.4 we observe that this lower execution time is accompanied by a decrease in operating frequency. The [32,32,32] configuration runs at 250 MHz which is almost 60 MHz higher than the 193.2 MHz of configuration [32,32,128]. We suspect that the increased RAM usage and constraints on local data decreases the operating frequency if not managed correctly. A full table containing all area and frequency specifications can be found in Appendix A.

The lowest execution time is achieved with the [16,32,64] configuration, which is clocked at 260.4 MHz. This speedup compared to the other low times can be attributed to a combination of the operating frequency and a higher throughput of Adx.

Table 6.4: Execution time, frequency and area of different unroll configurations for use-case (a)

back	BB^{T}	Adx	Exec(ms)	Freq(MHz)	%LUT	%FF	%RAM	%MLAB	%DSP
1	1	1	41.6	252.3	14.2	13.0	24.9	2.9	2.2
1	32	1	40.7	255.7	14.3	13.2	24.9	2.8	4.2
32	1	1	42.2	250	15.0	15.1	26.6	3.1	8.3
1	1	32	43.5	268.2	14.9	13.8	31.5	3.7	4.2
16	32	32	3.04	258.5	11.9	15.6	31.4	3.8	10.3
32	32	32	3.14	250	16.0	16.8	35.6	3.9	12.4
64	32	32	3.13	250	16.4	17.7	35.6	4.2	16.6
16	32	64	2.96	260.4	16.3	16.5	39.3	4.2	12.4
32	32	128	3.89	193.2	18.0	19.2	52.3	8.0	18.7

6.2.2. Double vs single precision floating-point data types

The reference data type used for the algorithm is double precision floating-point. The execution time of the algorithm on FPGA using double precision is 2.2 times longer compared to single precision. This difference is caused by an increase in latency and II of compute loops as double precision operations such as addition and multiplication take more cycles compared to single precision. For double precision operations multiple DSPs are chained together to provide the functionality, registers are used to buffer data between steps to not impact the clock frequency. Because of these registers multiple clock cycles are required before new data can enter the PE, which in turn can stall the pipeline and lower the II. Single precision operations only use one DSP and allow data to enter the PE every cycle, lowering the II of loops containing these instructions. Each CU uses one DSP in single precision and requires at least 4 DSPs for double precision, and this is excluding additional LUT and FF usage. Table 6.5 shows that limiting factor when adding CUs is not the number of DSPs or RAM available, rather it is the increased LUT usage. The double precision implementation performs reasonably well compared to the single precision implementation. Because latency of double precision operations is longer it impacts the loop latency and II of compute loops. Pipelining in and between the kernels hides parts of the latency allowing the double precision implementation to achieve only a 2.2 times slower execution time. The double precision implementation is currently using all the LUTs on the FPGA stopping any further optimizations. While for the single precision implementation there are still resources to continue optimizing.

Table 6.5: Area and frequency of single vs double precision configurations

sp/dp	back	BB^{T}	Adx	Freq(MHz)	% LUTs	% FFs	% RAMs	% MLABs	% DSPs
sp	1	1	1	252.3	14.2	13.0	24.9	2.9	2.2
sp	32	1	16	251.7	15.3	15.6	27.6	3.4	9.3
sp	16	32	32	258.5	11.9	15.6	31.4	3.8	10.3
dp	1	1	1	256.3	23.7	16.7	35.2	4.6	6.5
dp	32	1	16	223.5	83.3	40.0	40.6	13.1	35.0
dp	16	32	32	221.9	91.6	43.1	50.8	13.7	38.9

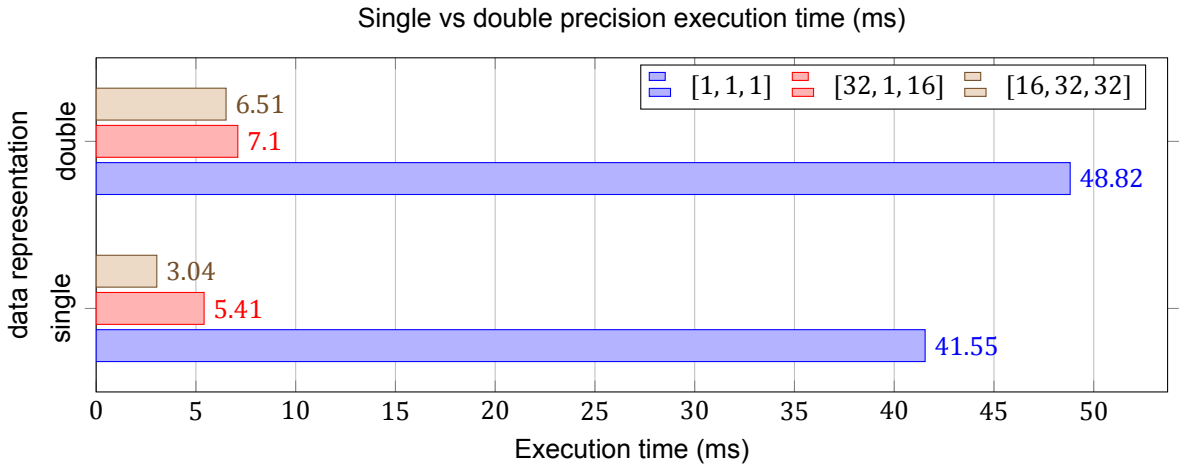

Figure 6.3: Single vs double precision execution time on FPGA, legend: [unroll backsub, unroll BB^{T}, unroll Adx]

6.2.3. Profiling the FPGA implementation

The Intel OpenCL SDK for FPGA is relatively new, the profiling tools are not yet mature. It does not manage to capture the complexity introduced by autorun kernels and channels. The compiler offers fast initial profiling by analyzing the OpenCL code before synthesis. The generated static report indicates how loops are scheduled and where the compiler could not unroll or pipeline kernels. Additionally, the user can see the estimated resources and memory structures used for all variables. With the information about the II and latency of loops, the user can already do some optimizations creating a fast design cycle.

Each bottleneck that arises can have many different solutions and causes. An alternative to the static reports from the SDK is to manually profile the application by adding counters and performance measurement signals to the kernels. This method is regularly used in traditional HDL and HLS languages to find bottlenecks in the system. In OpenCL kernels it becomes a complicated process to manually add counters, as there is no concept of a clock in a kernel and execution is sequential. To add these signals kernels have to be rewritten, this in turn changes the behavior and make the information less valuable. Instead, the ”-profile” flag of the compiler can be used and will add performance counters to the kernel. This adds significant overhead, Intel discourages using the profiler for kernels that run for less than 20ms. To analyze the data generated by these counters, the dynamic profiler provided by Intel can be used. This profiler provides information about the throughput and occupancy of memory and channels. It reports stall percentages about how many cycles are spend stalling the pipeline. These reports are straightforward for enqueued kernels as they have a well defined start and end. Autorun kernels start as soon as the device is programmed and stall on channel reads until data is present. Getting an accurate profile of autorun kernels is complicated, the host can request the data and reset the counters before and after execution of the controller. This improves the accuracy but because the algorithm is sequential kernels will have to wait most of the time until their inputs are ready. Viewing the exact interaction between kernels in the dynamic profiler is currently not feasible.

6.2.4. Further optimizations

There is still room in the implementation to improve, not just by solving pipeline stalls or adding more compute units. Making sure that all memory structures are correctly banked to the right number of read ports can further increase the frequency. Allowing us to extend the current unrolling without causing the frequency to drop.

The compiler currently chooses MLAB memory for small data structures that require multiple read ports. MLABs require a significant number ALMs while the ports might go unused because of regular access pattern. Manually banking all local memory structures to M20K blocks reduces the stress on routing and allows us implements only the required number of read ports.

The implementation is limited by the boundaries of the OpenCL framework, which might not be the best tool for this algorithm. As operations repeatedly use the same kind of operations, such as adding a constant to a vector. It makes sense to create a compute kernel for that specific operation. This kernel

is then fed from a multitude of kernels that use that computation. Currently, re-using on this level is not feasible, it is possible but at the cost of the expressiveness OpenCL provides.

6.3. Comparison FPGA, GPU and CPU

To find the best deployment strategy for the qpAS algorithm we compare our results on the FPGA to a comparable GPU and CPU. The reference implementation on CPU includes all of the necessary control operations and functionality whereas the results on FPGA and GPU focus mainly on the compute kernels. In Figure 6.4 we can see the execution time of all platforms for double precision floating-point and single precision for the FPGA. For both precisions we choose the fastest configuration. For single precision the [16,32,64] configuration is the fastest and for double precision the [16,32,32] configuration is the fastest. For double precision further increasing the unrolling factors is not possible, as the FPGA is already full.

The CPU performs best, regardless of the data representation, mostly due to the higher clock frequency that speeds up sequential execution compared to the 15x and 3x lower operation frequency of FPGA and GPU respectively. There are optimizations present in the FPGA implementation that can also be applied to the CPU version, but the library calls used are optimized by Intel for their systems. Most of the optimizations are not compatible with library calls, the operations used by these optimizations cannot be implemented using only general functions such as matrix multiplication. The accelerators use a separate memory system, as such the input data is sent from the host memory to the device through a PCIe link. From Table 6.3 only the dynamic data is included for the comparison, as the other data can be transferred outside of the critical path. In Figure 6.4 the transfer time required from the dynamic part is included for the FPGA. If DMA is used, transferring the static part takes 1.482 ms, while the dynamic part only takes 0.056 ms both for single precision data types. Double precision uses twice the number of bytes, this corresponds with the increase in transfer time of the dynamic part which grows to 0.111 ms. The static part does not fit in the aligned memory of host device, and is transferred without using DMA. This takes around 160 ms.

The lower execution time does not mean the FPGA implementation is not usable, the CPU and GPU both behave differently when multiple programs run in parallel. The best FPGA implementation does not use 100% of the resources yet, so multiple copies still fit on a single FPGA. Separate pipelines with dedicated interfaces to global memory can be used to make sure there are no dependencies between the copies. Assuming that the operating frequency can stay the same there is no performance loss with multiple copies of the algorithm running concurrently. On the CPU and GPU it is also possible to queue two copies of the algorithm, but as there is resource contention on the CPU it is not guaranteed that the execution time would remain the same. The MKL library functions use **Advanced Vector eXtensions** (AVX)2 or AVX-512 operations which can decrease the clock frequency. The lower frequency can impact the other copies, thus reducing the execution time. On the GPU the possibility of running multiple copies of the algorithm depends on the number of cores used by a single instance of the algorithm. In our use-case we use COTS libraries, the user has no control over how many cores and what block sizes are used. Guarantees in execution time can therefore not be made without testing over a multitude of data sets to ensure that deadlines are met.

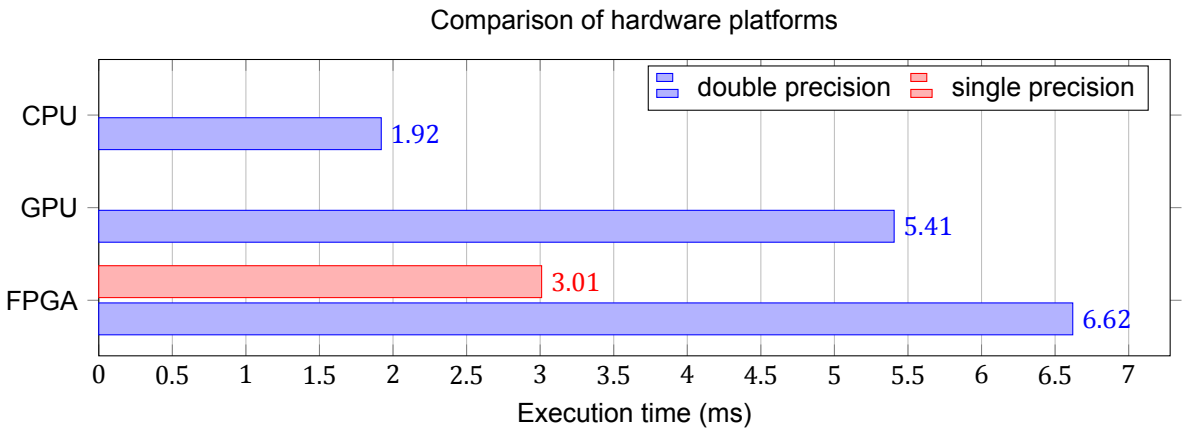

Figure 6.4: Execution + transfer time of the three hardware platforms for use-case (a)

6.4. Evaluation

The goal of this thesis is to explore the feasibility of deploying linear algebra algorithms on hardware accelerators using high-level languages. The results prove that with considerable effort it is possible to get competitive performance with an FPGA compared to a CPU or GPU implementation with COTS vendor optimized libraries. Separating read kernels and autorun compute kernels allows the user to create a self controlled system that requires little input from a host system to produce results. Using the FPGA to offload functions in the same way as using COTS libraries on a GPU will not deliver the same performance, as data streaming between functions is not possible. Converting the algorithm from sequential library calls to a custom kernel streaming implementation requires in-depth knowledge of the mathematics and underlying architecture.

In all three implementations there is still room for improvement. A significant optimization in the FPGA implementation is the use of rank-1 updates, which removes the need to compute full matrix-matrix or matrix-vector operations. However, COTS libraries do no provide the rank-1 update functionality. Therefore this optimization was not used in the CPU and GPU implementations. Currently, the COTS libraries do not contain efficient rank-1 update functions, always requiring full matrix-matrix or matrix-vector operations which can be reduced by efficiently processing the Active-Set. Using multiple cores on a high-end CPU provides the best performance without suffering from additional overhead such as launching applications and data transfers associated with using accelerators.

The data representation used in the reference implementation is not required for this particular use-case, more effort is needed to evaluate if all cases can use either mixed-precision or single precision. Using lower precision representations would increase the performance on both the CPU and GPU. For the OpenCL implementation there is still room to introduce more data streaming and reduce local memory requirements. This requires balanced kernels, each kernel must produce data at the same speed or faster than consumers can use it to prevent blocking the critical path. Even when an efficient implementation is achieved, the operation frequency of an FPGA is considerably lower than the frequency of a CPU.

7 Conclusion

In this thesis we explored acceleration of a linear algebra algorithm using OpenCL on an FPGA. The goal of the thesis was to get a better understanding of the OpenCL framework on FPGA and measure the performance for linear algebra use-cases which are regularly used in the computational models. In this chapter we answer our research questions and discuss possible future work on the subject.

7.1. Research questions

In this section we answer the questions proposed in the Introduction:

What parts of the algorithm are candidates for acceleration?

QpAS is an iterative algorithm that becomes more computationally complex as the problem grows in difficulty. In our analysis we found four major contributors to the execution time. The first is the Cholesky factorization over the Active-Set, which grows with each iteration of the algorithm. This growth is identified as the main bottleneck in the current CPU implementation. The second and third are the forward and backward substitution, these are simple but generally sequential operations making an impact on the execution time. The fourth is a relatively large matrix-vector multiplication that is in the critical path.

Can the algorithm be implemented on the FPGA using a high abstraction approach?

Using the OpenCL SDK for FPGA framework by Intel reduces the complexity of the development. Large algorithms that have a sequence of steps with a defined execution order can easily be implemented. However, a different mindset is required compared to developing for the CPU or GPU. Instead of splitting functions into kernels and implementing each part individually, a more dataflow oriented approach is required. The algorithm has to be converted to a streaming implementation before the development is started. Using the traditional approach of: kernel by kernel execution will not give performance on the FPGA.

A dataflow algorithm can be easily implemented using the features introduced by the OpenCL framework. Data can be streamed through channels that provide high bandwidth low latency interconnect between kernels. This data is then consumed by data triggered, explicitly synchronized autorun kernels. They will run as soon as data is available without instructions from a centralized controller. When all kernels in the pipeline consume and produce at the same rate, this approach generally has a high occupancy and thus high throughput.

The traditional approach of optimizing the kernels as black boxes and then calling those in succession, which is a common method to accelerate applications on the GPU and CPU, will not speed up the application on an FPGA. Developers have to adopt a dataflow oriented algorithm structure to match the underlying architecture. To implement an algorithm efficiently using streaming, experience and knowledge about FPGAs is required.

What are the performance trade-offs on the FPGA using the OpenCL framework by Intel?

Once an FPGA implementation is achieved, the next step is to optimize. One of the options is to unroll loops to add PEs to steps that are currently taking a long time. Only loops that have static bounds and perform independent operations can be unrolled. The new PEs increase the demands on the local memory structure. For large data structures, it is important to partition the data to provide the correct

port to the PEs. A regular access pattern eases the partitioning and implements only the required number of read ports. As the number of read ports on local memory (M20K blocks and MLABs) is limited, unrolling large loops that use data in local memory can be costly. It is important to only select the compute loops that are currently forming a bottleneck, and apply unrolling to increase the throughput. The DSP and logic usage will scale with the unroll factor. Additionally, the memory usage follows the same trend as each PE will require read ports. Partial unrolling factors can be used to limit the area usage for large loops.

Autorun kernels and channels are used to create a streaming based implementation of the algorithm. One of the restrictions on autorun kernels is that they cannot have a global memory interface. Channels are the only type of communication permitted. The default strategy is to have a read kernel constantly push data into channels connected to the autorun kernels. If the autorun kernel reads from a channel in an unrolled loop, it makes sense to duplicate the channel because parallel channels allow the autorun kernel to read more data in a single cycle at the cost of channel resources. Channels can have a configurable depth to buffer data if the consumer is not ready, but at some point the buffer is full. This is normal for read kernels as their only job is to provide data to an autorun kernel. If a channel connects two autorun kernels that are not balanced in throughput then one of the kernels will stall. This balance can be created by adding compute units to the lower throughput kernel or to reduce the resources used by the higher throughput kernel. It can be difficult to create this balance, as accurate profiling of the throughput is necessary.

Linear algebra algorithms typically use floating-point data representations. Using double precision is possible on the FPGA but drastically increases the resource usage. To get an efficient and high throughput implementation it is necessary to convert the algorithm to single precision. If double precision is absolutely required, moving to FPGA is not worth the effort.

What is the performance of the qpAS algorithm on the FPGA compared to a CPU and GPU using COTS libraries?

In this thesis we compare our FPGA implementation to a CPU and GPU implementation that both use COTS library functions. For the FPGA and GPU implementations we measure the execution time of the computational kernels and the required data transfer time. The CPU implementation also includes some additional control operations that have little impact on the execution time but are in the critical path. The algorithm based optimizations such as the rank-1 updates for the factorization and efficiently handling the Active-Set are not present in the CPU and GPU implementations. Our final FPGA implementation does use the aforementioned optimizations.

The execution time of the final FPGA implementation is 3.5x and 1.2x longer than the CPU and GPU respectively if double precision floating-point is used. If the precision of the FPGA implementation is reduced to single precision there is a speedup of 2.2x in execution time compared to double precision on the FPGA.

Our implementation is not optimal yet. Currently, the factorization and substitutions are still blocking speeding up the implementation further. Profiling the implementation to find the current bottlenecks and mitigating those to further balance the implementation will improve the performance. A handcrafted version of our implementation taking into account all the algorithm based optimizations could achieve even better results at the cost of flexibility and more development time. This implementation is out of scope for our thesis, which is limited to using a high abstraction level approach to implement the algorithm.

7.2. General remarks

During our work implementing the qpAS algorithm we recognize the benefits of using OpenCL kernels. Significantly less effort is required compared to hardware description languages to get to a working implementation. Most of the development time was spent on improving the performance by introducing streaming and unrolling loops.

The algorithm used determines the feasibility and performance of an OpenCL FPGA implementation. Not all algorithms are equally suitable to implement using a dataflow based approach. Most optimizations that proved valuable in reducing the computational complexity of our algorithm are not FPGA specific and could also be applied on the CPU and GPU. Additional effort spent on trying to accelerate a specific implementation must first focus on determining if the algorithm itself can be further optimized before considering alternative deployment platforms.

We believe that if a qpAS algorithm optimized implementation is created for all the considered hardware platforms the CPU will still achieve the best performance. The FPGA is expected to come in second because of the significantly lower clock frequency. The GPU is expected to benefit the least from the proposed algorithm specific optimizations as computational operations are exchanged for control and irregular execution.

7.3. Future work

The OpenCL framework for FPGA is a promising tool that allows developers with knowledge about the architecture to efficiently implement kernels with all required memory interfaces. The performance depends on the metrics used in the comparison. Purely looking at execution time of the algorithm on FPGA does not tell the whole story. Aspects that can be explored further include using the FPGA to compute multiple copies of the algorithm at the same time. The area usage of the final implementation in single precision allows us to fit at least 3 copies on the same FPGA. By duplicating the compute engines, the FPGA might have an advantage over the other platforms. Because each engine has its own datapath, there is limited influence on the rest of the FPGA in contrast to the CPU and GPU.

Further investigations are required on the effect of input size of the data, and the number of iterations required before the algorithm converges. Trade-offs between the number of PEs for each kernel can look differently depending on these sizes. There could be combinations of size and maximum number of iterations that benefits the FPGA more than other platforms, such as a smaller matrix-vector multiplication.

A common strategy when moving to FPGA offloading is changing the datatype from floating-point to fixed-point. The DSPs can do floating-point computations, but are often at the cost of clock cycles and a lower operating frequency. If we can move from a floating-point based algorithm to fixed-point then the FPGA will be a better fit. The datapath can be reduced to exactly the size required instead of 32- or 64-bit. The accuracy requirements of the output and input of the algorithm are unclear for our use-case. More research is required on the effect of reduced accuracy in the algorithm to allow the usage of fixed-point.

A

All FPGA configurations

In Table A.1 the execution time, operating frequency and area usage for all different unrolling configurations for the proposed implementation are given for reference. Most execution times can already be found in Figure 6.2.

Table A.1: Execution time, frequency and area of different unroll configurations for use-case (a)

sp/dp	back	BB^T	Adx	Exec(ms)	Freq(MHz)	%LUT	%FF	%RAM	%MLAB	%DSP
sp	1	1	1	41.6	252.3	14.2	13.0	24.9	2.9	2.2
sp	1	32	1	40.7	255.7	14.3	13.2	24.9	2.8	4.2
sp	32	1	1	42.2	250	15.0	15.1	26.6	3.1	8.3
sp	32	32	1	43.5	238.6	15.2	15.9	28.9	3.1	10.3
sp	64	1	1	43.3	241.7	15.8	17.0	28.9	3.8	12.5
sp	1	1	16	5.44	255.2	14.6	13.5	27.4	3.2	3.2
sp	1	32	16	5.09	252.3	14.6	13.8	27.4	3.1	5.2
sp	16	32	16	4.94	255.2	15.1	15.2	27.3	3.3	9.2
sp	32	1	16	5.41	251.7	15.3	15.6	27.6	3.4	9.3
sp	32	32	16	4.96	253.5	15.4	15.9	29.1	3.4	11.3
sp	1	1	32	43.5	268.2	14.9	13.8	31.5	3.7	4.2
sp	1	32	32	4.43	248.3	15.0	14.1	31.5	3.7	6.3
sp	16	32	32	3.04	258.5	11.9	15.6	31.4	3.8	10.3
sp	32	1	32	3.56	248.3	15.7	15.9	31.8	4.0	10.3
sp	32	32	32	3.14	250	16.0	16.8	35.6	3.9	12.4
sp	64	32	32	3.13	250	16.4	17.7	35.6	4.2	16.6
sp	1	1	64	4.88	244.1	15.8	14.5	37.2	5.0	6.5
sp	1	32	64	4.42	248.3	15.8	15.0	37.0	5.3	8.4
sp	16	32	64	2.96	260.4	16.3	16.5	39.3	4.2	12.4
sp	32	1	64	3.71	229.2	16.5	16.8	37.3	5.6	12.5
sp	32	32	64	3.24	235.8	16.6	17.2	38.7	5.5	14.5
sp	64	32	64	3.34	228.1	17.1	18.6	41.0	5.8	18.7
sp	1	1	128	5.81	202.7	17.1	16.7	50.6	7.8	10.5
sp	1	32	128	5.67	191.3	17.2	17.1	50.6	7.8	12.6
sp	16	32	128	3.59	211.8	17.7	18.5	50.5	7.9	16.6
sp	32	1	128	4.09	204.9	17.9	18.8	50.9	8.1	16.7
sp	32	32	128	3.9	193.2	18.0	19.2	52.3	8.0	18.7
dp	1	1	1	48.82	256.3	23.7	16.7	35.2	4.6	6.5
dp	32	1	16	7.10	223.5	83.3	40.0	40.6	13.1	35.0
dp	16	32	32	6.51	221.9	91.6	43.1	50.8	13.7	38.9

Bibliography

[1] Amazon AWS. EC2 F1 Instances, Sep 2019. URL https://aws.amazon.com/ec2/instance-types/f1.

[2] Mohamed A. Bamakhrama, Alejandro Arrizabalaga, Frank Overman, Jean-Paul Smeets, Kornel van der Sommen, Remko van der Vossen, and John Wagensveld. GPU acceleration of real-time control loops. *CoRR*, abs/1902.08018, 2019. URL http://arxiv.org/abs/1902.08018.

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13, Sep 2013. doi: 10.1145/2514740.

[4] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *2008 Symposium on Application Specific Processors*, pages 101–107, Jun 2008. doi: 10.1109/SASP.2008.4570793.

[5] J. Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 270–277, Dec 2013. doi: 10.1109/FPT.2013.6718365.

[6] J. Cong, Z. Fang, M. Huang, P. Wei, D. Wu, and C. H. Yu. Customizable computing—from single chip to datacenters. *Proceedings of the IEEE*, 107(1):185–203, Jan 2019. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2876372.

[7] Dell EMC. The importance of hardware raising all boats, 2018. URL https://education.dellemc.com/content/dam/dell-emc/documents/en-us/2018KS_Brant-The_Importance_of_Hardware_Raising_All_Boats.pdf.

[8] James Demmel. On floating point errors in cholesky. Technical report, Courant Institute, 1989.

[9] R. H. Dennard, F. H. Gaensslen, Hwa-Nien Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Proceedings of the IEEE*, 87(4):668–678, Apr 1999. ISSN 0018-9219. doi: 10.1109/JPROC.1999.752522.

[10] B. A. Draper, J. R. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe. Accelerated image processing on fpgas. *IEEE Transactions on Image Processing*, 12(12):1543–1551, Dec 2003. ISSN 1057-7149. doi: 10.1109/TIP.2003.819226.

[11] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, Oct 2019. ISSN 0949-877X. doi: 10.1007/s00778-019-00581-w. URL https://doi.org/10.1007/s00778-019-00581-w.

[12] N. Fujita, R. Kobayashi, Y. Yamaguchi, and T. Boku. Parallel processing on fpga combining computation and communication in opencl programming. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 479–488, May 2019. doi: 10.1109/IPDPSW.2019.00089.

[13] Azzam Haidar, Ahmad Abdelfatah, Stanimire Tomov, and Jack Dongarra. High-performance cholesky factorization for gpu-only execution. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 42–52, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4915-4. doi: 10.1145/3038228.3038237. URL http://doi.acm.org/10.1145/3038228.3038237.

[14] Joost Hoozemans, Rolf Heij, Jeroen van Straten, and Zaid Al-Ars. Vliw-based fpga computation fabric with streaming memory hierarchy for medical imaging applications. In *Applied Reconfigurable Computing*, pages 36–43, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56258-2.

[15] Joost Hoozemans, Jeroen Straten, Timo Viitanen, Aleksi Tervo, Jiri Kadlec, and Zaid Al-Ars. Almarvi execution platform: Heterogeneous video processing soc platform on fpga. *J. Signal Process. Syst.*, 91(1):61–73, Jan 2019. ISSN 1939-8018. doi: 10.1007/s11265-018-1424-1. URL https://doi.org/10.1007/s11265-018-1424-1.

[16] E. Houtgast, V. Sima, and Z. Al-Ars. High performance streaming smith-waterman implementation with implicit synchronization on intel fpga using opencl. In *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 492–496, Oct 2017. doi: 10.1109/BIBE.2017.000-6.

[17] R. Hunger. Floating point operations in matrix-vector calculus. Technical Report May, Technische Universität München, October 2005.

[18] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, Jul 2019. doi: 10.1109/IEEESTD.2019.8766229.

[19] Intel Corporation. Intel fpga sdk for opencl, Aug 2019. URL https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[20] Intel Corporation. Intel Arria 10 Product Table, Nov 2019. URL https://www.intel.co.uk/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf.

[21] Intel Corporation. Intel arria 10 core fabric and general purpose i/os handbook, Aug 2019. URL https://www.intel.com/content/www/us/en/programmable/documentation/sam1403483633377.html.

[22] Intel Corporation. Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide, Sep 2019. URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html.

[23] Intel Corporation. Intel Math Kernel Library, Nov 2019. URL https://software.intel.com/en-us/mkl.

[24] Intel Corporation. Intel Hyperflex FPGA Architecture, Nov 2019. URL https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10/features.html.

[25] Zheming Jin and Hal Finkel. Evaluating floating-point intensive applications on opencl FPGA platforms: A case study on the simplemoc kernel. In *2018 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2018, Cancun, Mexico, December 3-5, 2018*, pages 1–6, 2018. doi: 10.1109/RECONFIG.2018.8641693. URL https://doi.org/10.1109/RECONFIG.2018.8641693.

[26] Zheming Jin, Hal Finkel, Kazutomo Yoshii, and Franck Cappello. Evaluation of a floating-point intensive kernel on fpga. In *Euro-Par 2017: Parallel Processing Workshops*, pages 664–675, Cham, 2018. Springer International Publishing. ISBN 978-3-319-75178-8.

[27] C. Kachris and D. Soudris. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, Aug 2016. doi: 10.1109/FPL.2016.7577381.

[28] S. Kestur, J. D. Davis, and O. Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, Jul 2010. doi: 10.1109/ISVLSI.2010.84.

[29] Ryohei Kobayashi, Yuma Oobata, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. Opencl-ready high speed fpga network for reconfigurable high performance computing. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 192–201, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5372-4. doi: 10.1145/3149457.3149479. URL http://doi.acm.org/10.1145/3149457.3149479.

[30] K. Krommydas, A. E. Helal, A. Verma, and W. Feng. Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 198–198, May 2016. doi: 10.1109/FCCM.2016.56.

[31] Martin Langhammer and Bogdan Pasca. Floating-point dsp block architecture for fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 117–125, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3315-3. doi: 10.1145/2684746.2689071. URL http://doi.acm.org/10.1145/2684746.2689071.

[32] Martin Langhammer and Bogdan Pasca. High-performance qr decomposition for fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 183–188, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5614-5. doi: 10.1145/3174243.3174273. URL http://doi.acm.org/10.1145/3174243.3174273.

[33] Laurence Goasduff. Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020 , Aug 2019. URL https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io.

[34] S. Lee, J. Kim, and J. S. Vetter. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554, May 2016. doi: 10.1109/IPDPS.2016.28.

[35] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. FBLAS: streaming linear algebra on FPGA. *CoRR*, abs/1907.07929, 2019. URL http://arxiv.org/abs/1907.07929.

[36] Henry Megens. An introduction to photolithography: Overlay, 2007. URL https://staticwww.asml.com/doclib/productandservices/images/asml_overlay_images_fall07.pdf. Last accessed on: April 29, 2019.

[37] Microsoft Azure. What are field-programmable gate arrays (FPGA) and how to deploy, Sep 2019. URL https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-deploy-fpga-web-service.

[38] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11 (3):33–35, Sep 2006. ISSN 1098-4232. doi: 10.1109/N-SSC.2006.4785860.

[39] 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(10):1591–1604, Oct 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2513673.

[40] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, Mar 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.41.

[41] Nimbix, Inc. Accelerate your workflows with Xilinx Alveo™ Accelerator Cards in the Cloud, Sep 2019. URL https://www.nimbix.net/alveo/.

[42] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387303031. URL https://books.google.com/books?id=eNlPAAAAMAAJ.

[43] NVIDIA Corporation. CUDA, Sep 2019. URL https://developer.nvidia.com/cuda-toolkit.

[44] OpenACC-standard.org. OpenACC, Aug 2019. URL https://www.openacc.org/.

[45] 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)*, pages 270–277, Sep 2019. doi: 10.1109/FPL.2019.00051.

[46] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka. Evaluating high-level design strategies on fpgas for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep 2017. doi: 10.23919/FPL.2017.8056760.

[47] Sébastien Rousseaux, Damien Hubaux, Pierre Guisset, and Jean-Didier Legat. A high performance fpga-based accelerator for blas library implementation. In *Reconfigurable Systems Summer Institute 2007*, 2007.

[48] Ahmed Sanaullah and Martin C. Herbordt. Fpga hpc using opencl: Case study in 3d fft. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, HEART 2018, pages 7:1–7:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6542-0. doi: 10.1145/3241793.3241800. URL http://doi.acm.org/10.1145/3241793.3241800.

[49] Christian de Schryver. *FPGA Based Accelerators for Financial Applications*. Springer Publishing Company, Incorporated, 1st edition, 2015. ISBN 9783319154060.

[50] Valgrind Developers. Callgrind a call-graph generating cache and branch prediction profiler, Nov 2019. URL http://valgrind.org/docs/manual/cl-manual.html.

[51] Amulya Vishwanath. Enabling high-performance floating-point designs. White paper, Intel Corporation, 2016. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf.

[52] Z. Wang, B. He, and W. Zhang. A study of data partitioning on opencl-based fpgas. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep 2015. doi: 10.1109/FPL.2015.7293941.

[53] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on opencl-based fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, Aug 2016. doi: 10.1109/FPL.2016.7577329.

[54] Z. Wang, J. Paul, B. He, and W. Zhang. Multikernel data partitioning with channel on opencl-based fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1906–1918, Jun 2017. ISSN 1063-8210. doi: 10.1109/TVLSI.2017.2653818.

[55] Raymond J. Weber, Brock J. LaMeres, and Justin A. Hogan. Real-time , dynamic hardware accelerators for blas computation. In *International Journal on Recent and InNovation Trends in Computing and Communication*, 2017.

[56] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL http://doi.acm.org/10.1145/1498765.1498785.

[57] Xilinx Inc. SDAccel, Aug 2019. URL https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[58] Xilinx Inc. Vivado High-Level Synthesis, Sep 2019. URL https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[59] Xilinx Inc. MicroBlaze Soft Processor Core, Aug 2019. URL https://www.xilinx.com/products/design-tools/microblaze.html.

[60] F. Xu, H. Chen, X. Gong, and Q. Mei. Fast nonlinear model predictive control on fpga using particle swarm optimization. *IEEE Transactions on Industrial Electronics*, 63(1):310–321, Jan 2016. ISSN 1557-9948. doi: 10.1109/TIE.2015.2464171.

[61] Depeng Yang, Junqing Sun, JunKu Lee, Getao Liang, David D. Jenkins, Gregory D. Peterson, and Husheng Li. Performance comparison of cholesky decomposition on gpus and fpgas. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, 2010.

[62] M. Yih, J. M. Ota, J. D. Owens, and P. Muyan-Ozcelik. Fpga versus gpu for speed-limit-sign recognition. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 843–850, Nov 2018. doi: 10.1109/ITSC.2018.8569462.

[63] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, Nov 2016. doi: 10.1109/SC.2016.34.