UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

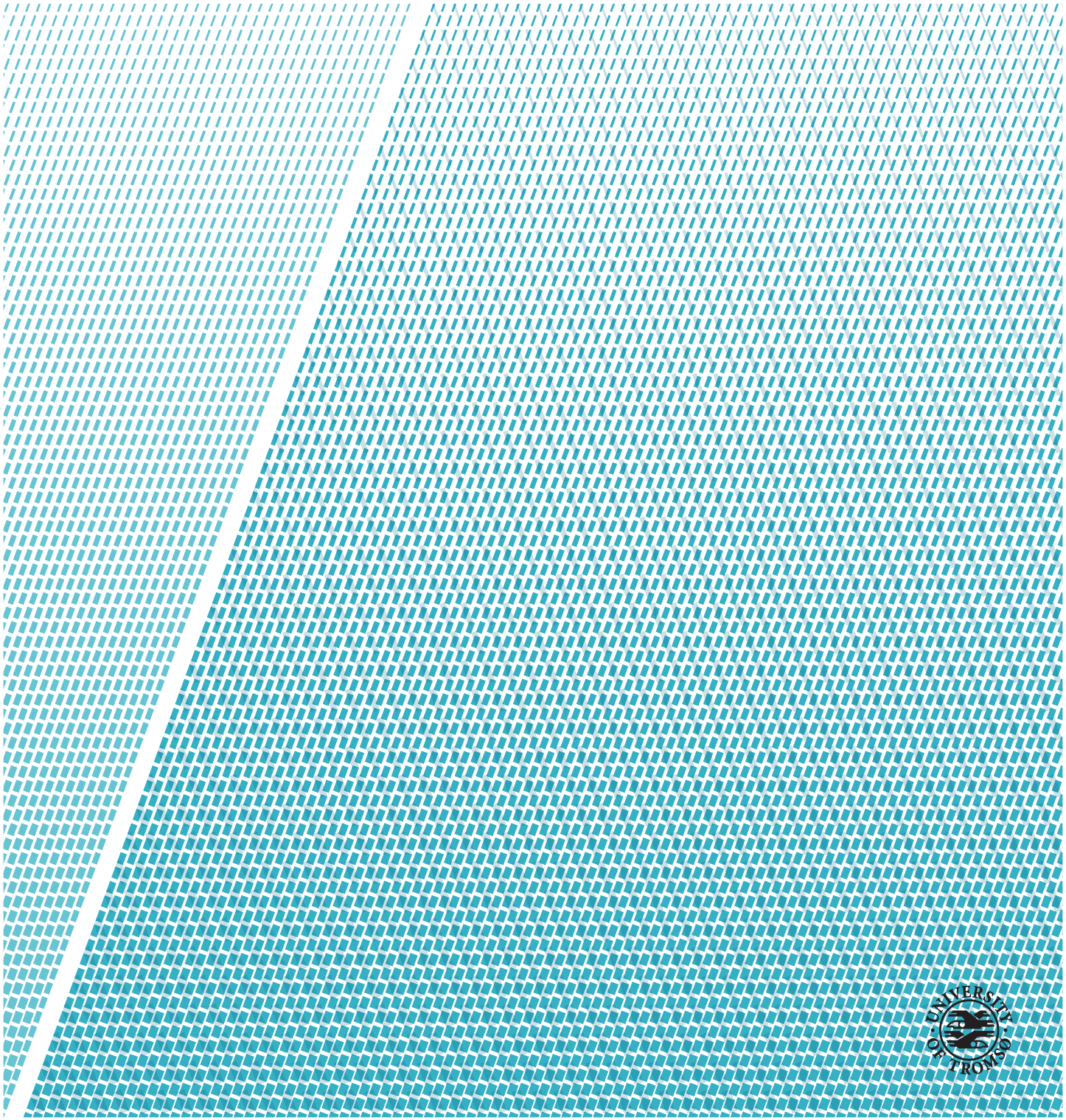
Useful GPGPU Programming Abstractions

A thorough analysis of GPGPU development frameworks

—

Johannes Larsen

Master thesis in Computer Science — June 2016



“Education is what remains after one has
forgotten what one has learned in school.”
–Albert Einstein

Abstract

Today, computers commonly have graphics hardware with a processing power far exceeding that of the main processors in the same machines. Modern graphics hardware consists of highly data-parallel processors, which are user programmable. However, software development utilizing these processors directly is reserved for platforms that require a fair bit of intimate knowledge about the underlying hardware architecture.

The need for specialized development platforms that expose the underlying parallelism to developers, elevates the learning threshold for newcomers, which obstructs the general adaption of GPU support. However, there are many frameworks that build upon, and elevate the abstraction level of, these specialized development platforms. These frameworks strive to provide programming interfaces less dependent on graphics architecture knowledge, and better resembling how you would program traditional software. They all come with their own quirks, and many of the abstractions they provide come with a considerable computational overhead.

This thesis aims to catalog relevant kinds of high-level GPGPU frameworks, and to evaluate their abstractions, and the overhead these abstraction impose on their applications. The experiments are based on real-world SAR processing problems that physicists at the university are exploring the possibility of accelerating on GPUs, and the experiments compare frameworks against each other and against a baseline low-level CUDA implementation. The results show that the overhead most frameworks impose are moderate for data-intensive problems, considerable for compute-intensive problems, and typically higher for high-level interpreted language bindings than for native frameworks.

Libraries with thoroughly tested general-purpose GPU functionality (e.g. ArrayFire) help in the development process, but must work on moderately sized data structures to amortize their overhead sufficiently. GPU code generators (e.g. VexCL) also have great potential, but their abstractions tend to add complexity to the code, which make them better suited for advanced GPU programmers, than regular developers.

Acknowledgements

First I would like to thank my main advisor, John Markus Bjørndalen for his support, and highly valued input to this project; my advisor at the physics department, Anthony Paul Doulgeris for providing the initial idea, benchmarking problems, and lots of help with the implementation details; and my external advisor at KSPT, Ole Morten Olsen, for being available, and willing to help, whenever problems arose.

Further I would like to thank KSPT for lending me the necessary hardware, and providing me with a nice office space. Along with my fellow student and office mate, Ruben Mæland for keeping me company this semester.

Furthermore, I would like to thank my fellow classmates and friends at the university for all these wonderful years. You will surely be missed!

And finally I wish to thank my family for encouraging me to do what I want, and for being supportive no matter what I choose to do.

Contents

List of Figures

List of Tables

List of Abbreviations

ADT Abstract Data Type

AMD Advanced Micro Devices, Inc.

API Application Programming Interface

ASIC Application Specific Integrated Circuit

BLAS Basic Linear Algebra Subprograms

CPU Central Processing Unit

CUB CUDA UnBound

CUDA Compute Unified Device Architecture

ECC Error-Correcting Code

EM Expectation-Maximization

EO Lab Earth Observation Laboratory

EOF End-Of-File

FPGA Field Programmable Gate Array

GPGPU General Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

GRD Ground Range Detected

HP Hewlett-Packard

HPC High Performance Computing

HSV Hue, Saturation and Value

I/O Input/Output

IC Integrated Circuit

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

JIT Just-In-Time

KSPT Kongsberg SPaceTec

MIC Many Integrated Core

MoG Mixture of Gaussian

MoGEM MoG Expectation-Maximization

MRCS Multi-variate Radar Cross Section

OpenACC Open Accelerators

OpenCL Open Compute Language

OpenMP Open Multi-Processing

PCIe Peripheral Component Interconnect Express

PGI the Portland Group, Inc.

RAII Resource Acquisition Is Initialization

R_{cr} CRoss-polarization Ratio

RGB Red, Green and Blue

RK Relative Kurtosis

ROM Read Only Memory

RTC Real-Time Computing

SAR Synthetic Aperture Radar

SAS Serial Attached SCSI

SATA Serial ATA

SIMD Single Instruction, Multiple Data

SNAP Sentinel Application Platform

SPMD Single Program, Multiple Data

SSD Solid State Drive

STL Standard Template Library

UiT Arctic University of Norway

VH Vertical transmit and Horizontal receive

VV Vertical transmit and Vertical receive

1 Introduction

Traditional graphics hardware used specialized processing pipelines for rendering graphics, but modern hardware has evolved into general-purpose highly data-parallel processors. Modern Graphics Processing Units (GPUs) are computational beasts, but lack of software support for making use of GPUs causes a general underutilization of their potential compute resources.

Using GPU hardware to compute non-graphics problems is the research field knowns as General Purpose computing on Graphics Processing Units (GPGPU). Today there are two major GPGPU platforms, Compute Unified Device Architecture (CUDA) and Open Compute Language (OpenCL), and each of them provide their own C-like interface to program GPU code. The interfaces these platforms provide are designed to facilitate performance critical application development, so they are intentionally exposing the underlying data parallelism to developers, which inherently complicates the interfaces.

Using these complicated programming interfaces elevates the learning threshold needed to develop software that utilizes GPU compute resources. Simplifying the development interface is one way lowering the initial learning threshold, and the common approach for doing so, involves the design of high-level abstractions intended to hide the underlying parallelism from developers.

1.1 Research Problem

The primary research problem for this master's thesis project is: *How can you best make abstractions for lowering the threshold needed to make use of the compute resources found in modern GPU hardware?* This thesis consists of the subproblems:

- What functionality does GPGPU abstractions provide, and how does this functionality benefit applications at large?
- What computational and programmable overhead do the usage of these abstractions inflict on the underlying application?

This research project is a collaboration between the computer science and the physics departments at Arctic University of Norway (UiT), and Kongsberg SPaceTec (KSPT).

1.2 Motivation

This project rose from the desire to optimize some of the time consuming satellite processing algorithms used by physicists at the Earth Observation Laboratory (EO Lab) at the UiT. These physicists were seeking guidance into the best approach for entering into the realm of GPU computing. They started with extensive MATLAB experience, and, given that background, they wanted to know what was the path of least resistance for getting a working GPU implementation, which would considerably outperform their current Central Processing Unit (CPU)-based implementation.

Given the specific algorithms they wanted to optimize, going straight for a CUDA or OpenCL implementation would most certainly provide something that outperformed their CPU implementation, but there surely exist frameworks that would simplify the development process. The exploration for such frameworks ensued, and this thesis is the result of the analysis to determine the optimal path forward for these physicists, and others in similar situations.

1.3 Contributions

The contributions of this work are:

- A thorough exploration of the high-level GPGPU frameworks that exists

within the CUDA development ecosystem.

- A comparison and analysis of the performance overhead imposed by the abstractions from a relevant subset of the researched GPGPU frameworks.

- An evaluation of the usefulness, programmability and pitfalls of using the abstractions provide by these frameworks.

- Implementations of some Synthetic Aperture Radar (SAR) processing and pattern recognition applications[1] in low-level CUDA code and various GPGPU frameworks.

1.4 Outline

Chapter 2 introduces the types, and uses, of computing accelerators, with a focus on GPUs, and it presents the evolution of the GPGPU research field.

Chapter 3 presents exploratory research into the capabilities and functionality of various types of high-level GPGPU frameworks.

Chapter 4 describes the benchmark experiments and their results, for evaluating and comparing, the functionality, and the performance overhead, of the most promising subset of the researched frameworks.

Chapter 5 analyzes the performance results, and evaluates the performance overhead of, and the usefulness of the abstractions provided by, the benchmarked frameworks.

Chapter 6 concludes this thesis, summarizes its findings, and outlines future work.

1. A chunk-wise statistical feature extractor (Section 4.2) and a terrain clustering procedure (Section 4.3).

2 Background

2.1 Computing accelerators

A dedicated computing accelerator is some form of specialized hardware designed to work in tandem with the main CPU. The hardware of such accelerators is designed to solve some subset of tasks, and doing them very well, as opposed to solving / being optimized for the assortment of tasks that is needed to be handled by the computer's CPU. GPUs are, by far, the most well known kind of accelerators, but those are not the only ones[7, I].

Many Integrated Core (MIC) is the common term for a processor architecture comprising lots of simple CPU cores connected together with a carefully considered communication interconnect among each other and their dedicated shared memory. This architecture is a middle ground between CPU and GPU design. One example of this architecture is the Intel's Xeon Phi, which is an extension card with in more than 50 x86-compatible processing cores with comprehensive Single Instruction, Multiple Data (SIMD) extensions that enables high-bandwidth[1] shared memory access. Their main selling point is the x86-compatibility, which means that they can be programmed like any regular CPUs, and they are designed to be direct competitors to GPUs in the High Performance Computing (HPC) community[10].

1. Intel® Xeon Phi™ Coprocessor 7120A: 352GB/s memory bandwidth, which is in the same ballpark as GPUs.

Field Programmable Gate Arrays (FPGAs) is a sort of reconfigurable Integrated Circuit (IC) that can be dynamically *programmed*[2] to form custom processing hardware.[7, III.A] Traditionally they and Application Specific Integrated Circuits (ASICs), their static counterparts, have commonly been used as coprocessors for signal processing and other sorts of real-time applications. FPGA development boards, which are regular extension cards, are nowadays being used more and more as general purpose streaming pipeline accelerators in applications where this sort of acceleration is deemed advantageous[7, I].

This thesis limits its scope to GPU hardware, but the other kinds of accelerators are mentioned because some of the frameworks are designed to write code that is agnostic to what sort of accelerator hardware it is supposed to utilize. This introduces a relevant factor in the choice of software framework, because the potential of having the possibility for utilizing other, or even future, kinds of processing hardware can prove to be a beneficial in the long run.

2.1.1 GPU Hardware

The graphics hardware is, at least traditionally, intended to off-load the CPU with graphical rendering support, which, in many scenarios, has some degree of Real-Time Computing (RTC) constraints. It is a piece of custom hardware that is optimized for solving the particular sorts of tasks associated with rendering graphics, and it usually has the physical hardware associated with handling all graphics up to, but not including the actual display. Some of the GPUs(e.g. [29]) are designed specifically for HPC applications, and can usually be distinguished from other kinds of GPUs by their lack of hardware for driving, and hence lack of connectors for, displays. That being said, their architecture are not that different from regular GPUs, and high-end regular consumer GPUs have reasonable performance, too[34, 1.1].

This subsection describes the hardware architecture of newer GPUs facilitated for general-purpose computing tasks. Earlier graphics hardware was targeted at a much more specific problem domain, so their design consisted of a static pipeline where each element had its own specific computation task.[39, A.1] Section 2.2.1 describes some of the reasons behind this hardware evolution, but apart from that, GPU in the context of this thesis always refers to hardware with general-purpose computing capabilities. The hardware description is based on, and references a description focusing on, NVIDIA GPUs, because the experiments and optimization discussion focuses on NVIDIA hardware, but the

2. FPGAs are bootstrapped with configuration parameters from a ROM, so the correct term is to *configure* their gates, but frameworks, such as OpenCL, provides programmers with a higher level of abstraction.

description is general enough to also apply to other kinds of GPU hardware with general-purpose computing capabilities.

Massive Multithreading

One of the main hardware design differences between CPU and GPU architectures is how they deal with the memory latency. CPUs use a hierarchy of large caches with orders of magnitudes lower latency than the main memory. These caches store the useful data, so that this particular data is readily available to the processor[39, 5.1]. GPUs, on the other hand, use an execution model similar to CPU hyperthreading, where a memory operation yields a context switch until the data becomes available, and, by storing enough concurrent contexts, overlaps the memory transfer delays with useful computing[39, A.4].

The caveat of how GPUs manage memory latency, is that it depends on the processor having enough concurrent computing tasks available to fill the waiting period of each individual execution context. Their solution is to scale up the multithreading, so current GPUs have hierarchies with multiprocessors totaling thousands of parallel processing units. Each of those have access to dozens of program contexts, which together make up enough useful computing work, so as to overlap their individual memory operations[39, A.4].

Execution Locality

If every program context were doing something completely different from every other program context, this would pose a major performance problem. For one thing, it would surely saturate the memory bus just by fetching everyone's next instruction. The approach in GPUs stems from the fact that these processor are designed to solve Single Program, Multiple Data (SPMD) problems, which means that bunches of threads are running the same program, but each having their own state.[39, p. A-22].

GPU threads are designed to be executed parallelly in bunches, and are statically, as in for their entire lifetime, bunched together in groups of 32[3] threads that are being executed in parallel. Diverging execution flows are discouraged, as this usually results in each distinct execution path being executed separately, while some subset of the other threads in the group are waiting idly[39, p. A-29].

3. This is suppose to be a device specific constant, *warp size*, but all CUDA compatible hardware has this fixed to 32, and this value is presumed by lots of software out there.

Having these blocks of threads executing in parallel influences another design choice where GPU hardware diverges from CPUs. The memory buses, and the cache lines (yes, GPUs have tiny memory caches), are much wider[4] than those on found on regular CPUs memory interfaces[34, 5.3.2]. The wide memory bus is intended to batch together accesses to neighboring memory locations required by separate threads within the same execution block into fewer aggregated memory operations. This batching amortizes the observed memory delay of an operation over the amount of threads that operate on data within the memory span of that operation, so every thread in a group accessing subsequent data elements at the same time will get their own element from the one and same memory operation. Whilst groups with threads accessing offset memory locations (e.g. the same column of subsequent rows in a row-major array) need one memory operation per thread.

The GPU memory cache hierarchy is organized in a similar fashion to, and it operates in the same manner as, its CPU counterpart, albeit having a much smaller capacity, but its objective is fundamentally different. CPU memory caches are designed for, and used to, hide the memory latency, but, as described in Section 2.1.1, GPUs have other means of approaching that problem, and, in any case, their workload tend to reuse data much less regularly than CPU applications[39, p. A-38]. Instead GPUs use their memory caches to maximize the memory bus utilization by minimizing redundant traffic[39, p. A-37].

2.2 GPGPU Platforms

The *G* in GPU stands for graphics, because that is the task that these accelerators were originally designed to perform. However, as described in Section 2.1.1, GPUs are simply processors optimized to solve massively parallelizable problems, and since the turn of this century there has been lots of research into how to exploit commodity graphics hardware into performing general purpose computation.

2.2.1 Evolution of GPGPU

The processing domain for commodity graphics hardware expanded when people started coming up with clever representations of much more complex graphical computation problems than those envisioned by the hardware design-

4. The GPU used in the experiment has a 320-bit memory interface[29, p. 2].

ers. An example of such work is [17], which computes Voronoi diagrams[5] using a GPU by computing distances as meshes of polygons. This trend continued with works mapping general purpose computation problems into the graphical domain, then mapping the resulting graphical solution back into the problem. The most novel of these works is [27], which computes matrix multiplication by mapping it intermediately as a graphics problem.

The earliest GPGPU research suffered inherent precision problems stemming from hardware limitations, but they postulated, correctly, that the hardware would evolve past those sorts of problems[27, 4]. Soon after, commodity graphics hardware evolved to support regular Institute of Electrical and Electronics Engineers (IEEE) floating point precision[26, 6], which provides deterministic rounding, and thereby results; and their programming interface changed to accommodate a more programmable shader[6] pipeline[26, 1], which gravely increased their application domain.

Programmable shaders were intended for extending the possibilities of graphical applications, so utilizing them for general purpose computation requires advanced graphics programming expertise[4, 1]. In an effort to making GPGPU more accessible to other kinds of developers, the field of GPGPU evolved frameworks providing an abstraction to the shader programming interfaces. One of the earliest example of such a framework is Brook[4], which provides an extended C language that compiles down to source code for the shader programming interfaces.

Vendor Specific Platforms

A couple of years later NVIDIA released their own proprietary GPGPU platform, which is vendor-specific, so it only works with NVIDIA hardware CUDA[34, 1.2]. It provides an extended C language (CUDA runtime) where you explicitly state whether a part is intended for the device (i.e. GPU) or the host (i.e. CPU). The CUDA runtime is built upon a low-level C-library(driver Application Programming Interface (API)), which provides the programmer with better control than the runtime API, but complicates the process of linking together host and device code.[34, 3].

At the same time Advanced Micro Devices, Inc. (AMD) released a beta version of their own low-level computation interface meant as a backend to the Brook compiler[2]. AMD's backend evolved to a GPGPU platform similar to CUDA, in

5. A space partitioning among a set of primitive geometrical shapes, such that each point belongs to its closest shape[17, 1].
6. Graphics filters that are expressed as arbitrary functions.

which the backend eventually have been dropped in favor of OpenCL[3].

OpenCL

OpenCL was released a couple of years after CUDA, and it is an open and vendor-independent programming interface for developing applications for heterogeneous parallel processing platforms. Where CUDA is optimized for, and limited to, programming GPU hardware, OpenCL is a much more ambitious project, which is intended to be used to programming various kinds of accelerator hardware.[22, 1]. The initial release defines an extended C-language very similar to the CUDA runtime for programming accelerator device(e.g. GPUs)[22, 6] kernels and a supporting C-library very similar to the CUDA driver API[22, 5]. Later releases extends the interface with support for workflows more suited for other kinds of accelerators (e.g. pipes for streaming processors[23, 5.4]), so its API is diverging from CUDA over time.

All the major GPU hardware manufactures provide proprietary OpenCL backends, albeit some of them are a bit outdated, so for anyone interested in the cross-platform aspect, OpenCL is the better choice of ecosystem. NVIDIA have had a head start since the release of CUDA, and CUDA code tends to a bit more optimized than equivalent OpenCL code on NVIDIA's hardware, but OpenCL is catching up, and given its openness and cross-platform aspect it will probably prevail in the end. The CUDA runtime abstractions makes it a fair bit easier to write GPGPU applications compared to using the OpenCL api, and application toolbox surrounding CUDA is superior when it comes to debugging, profiling and the like.

2.2.2 CUDA Kernels

Any code in CUDA that executes on a GPU is known as a kernel. These kernels are special entry point functions that the CPU schedules to be executed on the GPU. From the developer's perspective, scheduling a kernel execution looks more or less like calling a regular function, but in addition to regular function arguments, a set of special configuration parameters is also specified in the function call. The configuration parameters specify how the developer would like this code parallelly execute on the GPU, so the most important parameter concerns itself with how many threads that are suppose to execute that block of code in parallel.

The definition of such kernels looks like regular C/C++ functions, but in reality they declare a SPMD code block that will be executed in parallel. A kernel's function block can be thought of as the code block of a callable loop,

and in every kernel a special variable is available, which contains what would resemble the *loop counter* in this scenario. This *loop counter* is in reality an unique identifier of a specific thread of execution, and it is used as the index of data arrays just like loop counters in regular sequential loop constructs.

Kernel Fusion

```
# Think of each statement being compiled to a separate kernel
function dot(A,B)
        AB = A.*B # element-wise product
        sum(AB)
end

function dotFused(A,B)
        # map(f, A):    [a,...]          --> [f(a),...]
        # zip(A,B,...): [a,...], [b,...] --> [(a,b),...]
        sum(map((t)->t[1]*t[2], zip(A,B)))
end
```

Listing 2.1: Pseudocode kernel fusion example. Kernel: $A \cdot B = \sum_{i=0}^{N} A_i B_i$.

Listing 2.1 visualizes a common optimization technique known as kernel fusion. The procedure for doing this takes general compute kernels (e.g. element-wise product) and merges them into a single kernel. The idea is that there is a more or less fixed, but far from neglectable, overhead of executing any one kernel, so merging more than one operation into a single kernel results in less aggregated execution overhead than if all these operations were implemented as separate kernels.

3

High-Level GPGPU Frameworks

As described in Section 2.2.1, GPGPU has evolved from a field reserved for graphics gurus to spawning frameworks that makes programming them fairly comparable to programming regular CPUs. GPUs are parallel processors, and, since their platforms expose the parallelism, programming them inherently requires concurrent programming knowledge. The explicit thread orchestration described in Section 2.2.2 and isolated memory space[1] are additional sources for errors, but apart from that, getting to a working program is little different from writing regular C code. Optimizing the program for running on GPU hardware requires a little more intimate knowledge about how GPUs work, but this is getting easier by each platform generation, and today you get a long way with just as much knowledge as described in Section 2.1.1.

The GPGPU platforms have come a long way, but their programming interface is still fairly low-level, and, since many developers prefer to work at a higher level of abstraction, many general-purpose libraries and alternate programming interfaces have emerged. The motivation behind these frameworks differs slightly, but their main goal is usually to somehow simplify the transition to GPU software development. This chapter presents, and tries to categorize, a somewhat

1. Newer framework versions supports somewhat automatically synchronized memory[34, 3.2.7][23, 3.3.3].

relevant portion of the available frameworks. The following sections describe the framework categories, and Table 3.2 (end of chapter) gives a summarized overview. The categories are not mutually exclusive, so frameworks providing functionality spanning more than one category are presented in all of them, but the different descriptions focus on separate parts of the frameworks.

This thesis focuses on frameworks from the ecosystem around CUDA, rather than OpenCL, since that is the most mature of the two. This choice is intended to give the frameworks the best fighting chance in the experiments, but in the framework description I mention similar OpenCL abstractions whenever they exist. Some of these frameworks have additional support for seamlessly utilizing multiple GPUs, but this thesis limits its scope to single-GPU applications.

The names of CUDA/OpenCL frameworks and libraries tend to being pre- or suffixed with *cu/cl*, so in those particular cases it has not been emphasized whether they are CUDA or OpenCL frameworks. In general OpenCL frameworks tend to be more free then the official CUDA libraries, both in the sense that a much greater share of them are open-source, but also with respect to the fact that there is a much more commercialized ecosystem surrounding CUDA. There is also something fundamentally different about CUDA libraries being hand-optimized to their GPU architecture and OpenCL libraries being written to support a much greater range of hardware. Some OpenCL libraries mitigate this by using Just-In-Time (JIT)-compiled hand-optimized kernels chosen based on performance heuristic, and many others are simply designed oblivious to the fact that they may run on non-GPU hardware.

3.1 Domain Specific Libraries

CUDA	OpenCL	Description
cuBLAS	clBLAS	Basic Linear Algebra Subprograms, a well known vector/matrix math library.
cuFFT	clFFT	Fast Fourier Transform, some complex signal processing algorithms.
cuSPARSE	clSPARSE	Partial BLAS functionality for sparse matrices.
cuDNN	jtorch,. . .	Deep Neural Network functionality. Torch is one sort, so OpenCL has more specific libraries.
OpenCV		Computer Vision functionality.
NPP	OpenCLIPP	Integrated Performance Primitives, image and signal processing primitives.

Table 3.1: Examples of domain specific GPU libraries.

First of all, let us emphasize the fact that there exists lots of special-purpose GPU libraries, so applications that deal with common problems might benefit from surveying the ecosystems for libraries that completely, or at least partially, solve their problem. Some of the most well-known libraries are summarized in Table 3.1, to give examples of domains for which these kinds of libraries exist, but this thesis focuses on general-purpose abstractions, so those not relevant for the specific benchmark will not be discussed any further.

3.2 Kernel Code Generators

Let us start this survey with a fairly low-level set of abstractions, and continue with increasingly higher-level abstraction frameworks. The frameworks described in this section provide various kinds of reusable general-purpose functionality to simplify the act of writing kernel code.

First of all, CUDA has support for using C++ templates and lambdas in their kernel code, which in itself gives it an advantage when it comes to writing reusable code. On the other hand, OpenCL JIT-compiles the kernels, which, in spite of adding a bit of initialization overhead, gives them an opportunity to generate code that is even better optimized to the particular problem.

3.2.1 CUB

CUDA UnBound (CUB)[36] is a CUDA template library that provides primitives for doing common tasks in parallel without bothering with the details of how to optimize that sort of parallelization for the particular hardware it is supposed to run on. The primitives range from generic reduce and sorting functionality that is used to create custom kernels, to generically generated multi-pass kernels for solving common tasks. The motivation behind CUB is to provide parallelizable building blocks that ensure correctness and provide the developer with the optimum choice of underlying implementation.

Newer GPU hardware have better compute capabilities[2], so the optimum implementation for some functionality might depend on what hardware features are supported by the GPUs running the code. Abstracting away how these sort of primitives are implemented also opens for the possibility of supporting future optimization without the need for reimplementing the code. CUB also provides alternative algorithmic variants of the different primitives, in order

2. E.g. NVIDIA compute capability 3.0 introduced warp shuffling, which supports intra-warp reductions without need for shared memory[34, B.14].

for developers to simply switch between different implementations and test which one results in the best performances[36, 4].

OpenCL does not have anything as substantial as CUB, and its closest competitor is the SkelCL[43] library. This library provides skeleton C++ templates for common generic algorithms (e.g. map and reduce) with a user-supplied OpenCL code representing the lambda function for the actual operation.

3.2.2 Hemi

Hemi[16] is a another CUDA template library trying to simplify hybrid CPU/GPU development. Its main contribution is a layer of abstraction to the relevant CUDA C-extensions, so that the code can be compiled to either generate sequential CPU code or automatically orchestrated parallel GPU code. Additionally it provides a nifty parallel-for construct with, albeit experimental, native C++ lambda support, and some simplified orchestration and memory management directives.[16, README.md].

3.2.3 Kernel Fusers

Section 2.2.2 describes a common technique for optimizing GPU code. The interface for doing this sort of merging procedure, and the functionality supported, differs among the frameworks, but the underlying idea is the same.

The first framework that utilized this abstract method of generating complex kernels is the Standard Template Library (STL)[3]-like library, Thrust[35], for CUDA (later also Bolt[1] for OpenCL). This library provides a set of algorithms (e.g. map, reduce and sort) which constitutes basic kernels, and any of these are executed separately. The algorithms operate on data in the form of generic iterators, so any algorithm can be fused with arbitrary element-wise operations by wrapping the data accessors that arbitrarily transform the data[35, 3]. Listing 2.1 tries to illustrate the concept, but it should be noted that the C++ syntax for creating transformation iterators and the like is quite a bit more verbose.

VexCL[11] is a multi-platform C++ vector expression library that automatically fuses expressions into combined compute kernels. These expressions behave like numerical types, so they yield considerable better readability, but it does require runtime JIT-compilation of the expressions, as opposed to template

3. The collection-like part of the standard C++ library, which provides abstract data containers with generic content and general algorithms using such containers.

libraries that determine the combined expression at compile-time. The library lso supports more complicated functionality with custom kernel generators, so it can ultimately be just as generic and has at least the same expressiveness as Thrust and the like.

Tensor libraries and other sorts of expression optimization frameworks are also heavily devoted to kernel fusion, but their process is more automatic and they seldom provide any user-interface to manually organize how smaller kernels are supposed to be combined.

3.3 Container Data Type

The previous section briefly touched upon the topic of abstract data interfaces, which brings us to the next category of frameworks. This section describe frameworks that provide Abstract Data Type (ADT) interfaces to containers for data residing in the GPU memory space. The idea behind these sorts of frameworks is to work on data residing in the GPU memory as if it was a regular data type, without transferring intermediate results back to the main memory.

3.3.1 `std::vector`

The simplest data container explored by this thesis is GPU vectors (i.e. flat indexable arrays). CUDA's Thrust, OpenCL's Bolt and SkelCL provide an interface to a GPU vector that mimics the C++ `std::vector` data type[35, 2][11, Vector]. These vectors can be allocated/initialized and copied back as regular C++ vectors, and the frameworks support user customizable constructs for primitives (e.g. map and reduce) that access these vectors. The C++ vectors are abstract data containers, so they themselves are simply oblivious to their actual content, and, as such, have very intricate interfaces for expressing operations on their actual data.

Other frameworks, such as VexCL, limit their vectors to only support numerical data types, and they therefore have the ability to provide much more extensive interfaces. Having interfaces that mimics mathematical expressions greatly simplifies how to access the underlying data, and, as such, makes for much more readable code, but it does this at the cost of the generality of the data container.

3.3.2 Multi-Dimensional Arrays

When frameworks choose to impose mathematical interfaces on their data containers, the next logical step is multidimensional array (i.e. matrices). The generality of C++ vector iterators makes it possible to map more complicated structure (e.g. a matrix) onto the underlying data vectors in frameworks like Thrust, but this merely complicates the process of operating on the underlying data and the level of indirection decreases performance, so, although possible, these frameworks are not really suited for this kind of data representation. There are, however, lots of frameworks that focus specifically on providing a useful multi-dimensional array interfaces to the user.

Basic Linear Algebra Subprograms (BLAS) libraries are the first that come to mind, and they provide functionality for operating on arrays as vector and/or matrices, so cu-/clBLAS can be said to to support multi-dimensional arrays, although they simply work with data pointers and do not by themselves provide any specific data container type. VexCL and SkelCL provides specific container types for matrices[11, primitives.html][43, Matrix], but these seems to support fairly little functionality, so they are probably works in progress and/or proof of concepts.

ArrayFire

ArrayFire[45] is a framework focused around its 4-dimensional array container object, which has an extensive library surrounding it. It is primarily a C/C++ library, but there also exists, at least as works in progress, a lot of alternate language bindings. As with VexCL the framework is multi-platform, and it has the possibility of switching between CUDA, OpenCL and multithreaded CPU backends either compile- or runtime, which in itself makes this an attractive framework for research purposes, because it opens for the possibility of comparing cross-platform performance of the same application[45, README.md].

The interface is designed for passing around materialized arrays, so it is very similar to how you would program using a language like MATLAB. The similarities are probably not a coincidence, given the fact that it has evolved alongside a MATLAB GPU extension, and that the people behind it are, or at least have been, involved with the development of the official MATLAB GPU support[21].

The ArrayFire library has functionality spanning many mathematical domains, and, at least when it comes to extent of implemented functionality, it seems to be, as far as I have come across, the most comprehensive open-source GPU library. It has some support for automatically fusing simple functions (e.g. C++ mathematical operators on the arrays) into JIT-compiled special-

purpose kernels, and it has a construct for vectorizing[4] more or less arbitrary expressions, but most of the functionality consists of pre-programmed kernels made available to the developers via a nice API.

3.3.3 Tensors

A tensor is a symbolic representation of a mathematical expression, and the following frameworks provide data types that when used in expressions represent a symbolic parse tree of their combined operations. At some point in the program these tensor are associated with input variables and lazily-built/evaluated to produce the output variables. The main advantage to this sort of approach is that providing the framework with a symbolic representation of the algorithm gives it a greater opportunity to optimize the mathematical expression (e.g. $x = \frac{yz}{y} = \frac{\not{y}z}{\not{y}} = z$) than if it were interpreted imperatively.

Theano[44] is a Python framework and mshadow[8] is a C++ framework that provides symbolic tensor data structures, and evaluators. Theano is a library for an interpreted language, so you explicitly build a kernel for evaluating some given set of expressions at some point in your code, then you evaluate this as a function over some given input data[44, II] to compute its output. The tensors in mshadow, on the other hand, are programmed in a compiled language, and they utilize C++ Expression Templates to optimize the expression into fused kernels at compile time[8, guide]. Afterwards these kernels are use to compute the expressions at run time. The expression templates in VexCL, which are described briefly in Section 3.2.3, provides a similar sort of interface, but these expressions are interpreted more explicitly, so there is no mathematical optimization of the expressions, and they are expressing mutations directly on data objects.

3.4 Abstract Parallelizers

Open Multi-Processing (OpenMP)[38] is a set of compiler directives (C/C++/Fortran) in widespread us that provides a portable method for describing the high-level aspects relating to SPMD code parallelization[38, 1.0]. Newer versions of OpenMP have added support for offloading the parallelized computation to dedicated accelerator hardware[38, D.2], but mainstream compiler support for these features is still in the development phase. Open Accelerators (OpenACC)[37] is similar set of compiler directives, which have a greater focus on the aspects of offloading the parallelization, and these directives are de-

4. Convert element-wise statements to operate on an entire vector in parallel.

clared a bit more open-ended than the equivalent OpenMP directives, which gives the compiler more leeway in its choice of how to parallelize the code[37, 1.0]. As with OpenMP computation offloading, mainstream compiler support for OpenACC is still in the development phase. However, there are proprietary compilers, such as the Portland Group, Inc. (PGI), that have extensive OpenACC support already.

The idea behind these sort of compiler directives is that developers starts with the source code for a working sequential application, then they annotate the orchestration parts (e.g. loop constructs) of the source code with directives that describes how the code should be parallelized.. These annotations are designed to be optional to the compiler, so if they are simply ignored, either by a compiler oblivious to their meaning, or by having their support disabled, the compilation yields the original sequential program.

OpenMP constructs are designed to give users the ability to dictate explicitly how the compiler is suppose to parallelize the code, and subsequently how that would be achievable (e.g. where it would need synchronization). This approach opens for the possibility of parallelizing applications that are sequential in nature, but, by specifying how to orchestrate the parallelization, it may require different approaches to suit different parallelization architectures. OpenACC, on the other hand, takes a more lenient approach. Its annotations are meant more as guiding information about the developer's parallelization intentions. This shifts more of the responsibility for how to parallelize the code over to the compiler, which avoids the need for differentiating the parallelization scheme for separate architectures. Whereas OpenMP provides explicit constructs for handling synchronization, having similar functionality in OpenACC would undermine the compiler's choices for parallelization. Instead OpenACC offers constructs for indicating data dependency information, and limits the scope of the original applications to those that developers have already made inherently parallelizable (e.g. no data races).

3.5 Language Bindings

Some people prefer to develop in non-C/C++ environments, so dozens of language bindings for doing GPU development have been developed over the years. I have come across low-level CUDA and/or OpenCL interface ports to languages such as, among others, FORTRAN[34, 1.2], Haskell[6], java[19], Julia[18], MATLAB[28] and Python[24]. There seems to have been a bit more development effort behind the language bindings for CUDA than for OpenCL, but there is a substantial diversity in both ecosystems.

Simply porting the low-level interface gives developers the possibility of working in other languages, but many of these ports simply provide an interface to that which resembles the CUDA driver API, and still requires developers to write native CUDA/OpenCL kernel code. Other provides higher-level interfaces (usually as an addition to the low-level driver API) that are better suited for the language in mind and/or somehow tries to simplify GPU development.

PyCUDA/PyOpenCL provides a GPU container mimicking the NumPy's *n*-dimensional array structure, but it requires additional, and mysteriously complex, libraries for supporting more than element- and vector-wise functionality. The MATLAB Parallel Computing Toolbox provides a GPU array container data type, which works, in most cases (i.e. supported by in excess of 300 functions[28, Run Built-In Functions on a GPU]), as a drop-in replacement for regular MATLAB arrays. The aforementioned Haskell bindings only have an interface resembling tensors or VexCL much more than the original low-level API, and that is understandable given the language's lack of imperative programming constructs.

An alternative to creating language specific-extensions is to provide language bindings on top of higher-level frameworks. ArrayFire, for instance, has, in addition to its native C/C++ interface, stable bindings for Python and Rust, and they are working on bindings for a lot of other languages[45, README.md]. There are also many language bindings to the most common domain-specific libraries (e.g. [19, JCublas]), but, if they are not reimplemented from scratch, they, as their corresponding low-level language bindings, tend to be verbatim copies of the low-level interfaces.

CUDA	OpenCL	License	Language	Description
CUDAfy.NET		LGPL2.1	C#	Low-level
-	cl4d	Boost	D	Low-level
CUDA Fortran	-	Proprietary	Fortran	Low-level
-	FortranCL	GPL	Fortran	Low-level
cuda5	go-opencl	?	Go	Low-level
jcuda	jocl	MIT/X11	Java	Low-level
-	WebCL[a]	N/A	JavaScript	Low-level
CUDArt.jl	OpenCL.jl	MIT	Julia	Low-level[b]
CUDALink	OpenCLLink	Proprietary	Matematica	OpenCL:low-level, CUDA:BLAS1[d]
CUDA-minimal	-	Perl	Perl	Low-level
Thrust	-	Proprietary	C++	Vector[c], BLAS1[d]
-	Bolt	Apache	C++	Vector[c], BLAS1[d]
-	Boost.Compute	Boost	C++	Vector[c], BLAS1[d]
-	SkelCL	MIT	C++	2D-matrix, BLAS1[d]
PyCUDA	PyOpenCL	MIT	Python	*n*D NumPy array, BLAS1[d]
VexCL		MIT	C++	Vector+sparse 2D matrix, BLAS2[e]
ArrayFire		BSD	C,C++,...	4D array, BLAS3[f]++
PCT[g]	-	Proprietary	MATLAB	*n*D array, BLAS3[f]++
mshadow	-	Apache	C++	2D tensor, BLAS3[f]
Theano	-	BSD	Python	4D tensor, BLAS3[f]
accelerate		BSD	Haskell	*n*D tensor, BLAS1[d]
CUB	-	New BSD	CUDA C++	Reusable kernel patterns
Hemi	-	BSD	CUDA C++	Portable kernel patterns
OpenMP		N/A	C,C++,Fortran	Specific parallelization pragmas
OpenACC		N/A	C,C++,Fortran	Abstract parallelization pragmas

a. No major browser or any other kind of JavaScript runtime environment supports it.
b. OpenCL version has higher ambitions of mimicking PyOpenCL, but that is work in progress.
c. Custom iterators can map more complex structure, but this gets complicated quickly.
d. BLAS level 1: vector operations, so elementwise and vector reductions (e.g. dot product).
e. BLAS level 2: Matrix-vector operations, so row-/colwise reductions (e.g. matrix dot vector).
f. BLAS level 3: Matrix-matrix operations, so matrix products and solve for matrix
g. Parallel Computing Toolbox

Table 3.2: Summary of high-level GPGPU frameworks

4

Experiment

This project does not have the resources to explore neither every application domain nor every optimization framework, which yields a trade-off between the experiment quality with respect to application generality and framework representation. In order to avoid leaving out important optimization frameworks, the experiment focuses on a few fairly general-purpose GPU processing schemes that span many accelerator usage domains, so the choice has been made to prioritize exploring framework diversity over how thoroughly each framework is benchmarked.

Choosing how to best represent a wide range of application domains with few benchmarks requires the different benchmarks to stress fundamentally different usage scenarios. One of the major bottlenecks, if not the most important, when it comes to GPU acceleration is memory access, so it is obvious that the different benchmarks should try to cover different memory access patterns. The external memory bus is, at least, an order of magnitude slower than the internal GPU memory speed, so it is very important to minimize the amount of data transfer between the GPU memory and other resources[1]. However, there are lots of applications where a high amount of external memory transfer is inherently unavoidable. The most common of these is any application that works on a dataset much larger than what can fit within the limited memory space of the GPU, which requires them to work on smaller chunks one at a

1. The most common external resource is CPU memory, but memory of separate GPUs and other I/O-devices are also relevant.

time. This sort of processing scheme is known as a streaming pipeline, and is the basis for the benchmark described in Section 4.2.

The pipeline approach, although relevant, has the inherent problem that the memory transfers tend to mask differences in computing performance, which makes it unsuitable for a benchmarking compute intensive applications. Therefore, as a means of representing a wider range of applications, an alternate benchmark taking the opposite approach has been devised. This benchmark, which is described in Section 4.3, reuses the same data over and over again during its computations, which yields a fairly fixed memory footprint, and by limiting the workload, such that the entire problem fits within the GPU's memory, unnecessary (i.e. all but transferring input arguments and results from/to CPU) memory transfers are avoided.

Together these benchmarks are supposed to represent a great deal of general purpose problems that are commonly accelerated by utilizing GPU computation resources. Both benchmarks are based on real problems that physicists at the university are exploring the possibility of accelerating on GPUs, so the implementation obstacles presented are real, and therefore relevant to discussion about the framework interface differences. The physicists kindly provided CPU-based reference implementations written in MATLAB, with no regard for how to optimize the problems for GPU computation. The following sections presents simplified summaries of the problem domain for these benchmarks. More information can be found in [13, 2](algorithmic overview) and [12](statistical analysis).

4.1 Experimental Setup

The experiments are conducted on a Hewlett-Packard (HP) Z820 workstation with a NVIDIA Tesla K20 HPC GPU. The workstation has dual octa-core Intel CPUs[2] and 32GB of Error-Correcting Code (ECC) system memory[3].

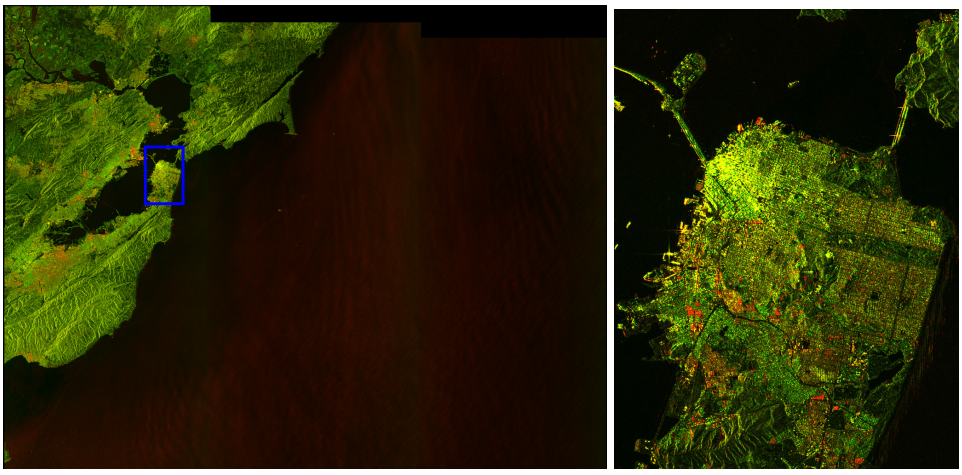
The GPU is a Peripheral Component Interconnect Express (PCIe) Gen2 x16 extension device with a single GPU[4] and 4.8GB ECC memory. It has a theoretical 208GB/s intra-device memory rate[29, p. 2], and its theoretically 8GB/s[40, p. 38] PCIe bus rate is the bottleneck for inter-device (including system memory) communication. The system drive is a 256GB Solid State Drive (SSD)[5]

2. Two Intel Xeon E5-2690 (2.9GHz) octa-core CPUs with hyperthreading disabled.
3. Two 8GB HMT31GR7EFR4C-PB DDR3 1600MHz memory sticks per CPU socket.
4. A GK110 with 2496 thread processors at 706MHz[29, p.12]
5. A 256GB Micron MTFDDAK256MAM SSD.

connected via a Serial ATA (SATA) 6Gb/s interface[6].

4.1.1 Dataset

This project uses satellite data from the ESA Copernicus project, which is freely[7] available through their data portal. The data used in this project are SAR scenes from the Sentinel 1 mission with dual-co/cross polarization. SAR records intensity and phase of reflected radar waves and represent this as complex numbers[15, 1.1], but various GPU frameworks have inadequate complex number support, so this project limits its data range to Ground Range Detected (GRD) products, because those only contain intensity data (i.e. real numbers).



(a) Entire scene (Section 4.2). 25289×19431

(b) Cropped to San Fransisco (Section 4.3). 1500×2000

Figure 4.1: Satellite scene over San Fransisco used throughout the experiments. This particular scene is flipped horizontally, but that is simply a natural effect of the satellite's acquisition path, and is totally irrelevant to the benchmark. Red:VV, green:VH, and polarizations have been normalized around their respective means separately. Copernicus Sentinel data 2016.

Figure 4.1 visualizes a satellite SAR scene[8] overlooking the San Fransisco Bay Area, and this scene is used as the dataset throughout the following experiments. The entire scene is about half a gigapixel in size, which, represented as single precision floating points, yields an almost 2GB large matrix per polarization.

6. A port on the integrated LSI SAS2308 SAS controller.
7. Sentinel Data license resembles CC-BY-3.0, see `https://sentinel.esa.int/documents/247904/690755/Sentinel_Data_Legal_Notice` for details.
8. `S1A_IW_GRDH_1SDV_20160429T141541_20160429T141610_011037_01099F_6340`

The pixel data used by the experiments have been radiometrically calibrated using ESA's Sentinel Application Platform (SNAP) toolbox, and converted[9] to files storing memory dumps of column-major matrices.

4.1.2 Intermediate Files

The benchmarks span language domains and are designed to be decoupled, so, for testability and making smaller parts of the benchmarks comparable, data is read and, optionally, written to the filesystem. This decoupling adds a bit of overhead when compared to an application that does everything in one go, but, as these benchmarks are not supposed to be run together, the impact to the overall runtime is ignored.

MATLAB stores variables as a compressed key-value store[28, `save()`], ArrayFire has a similar file format[45, `af_save_array()`] and most simple raw image format stores data in a row-major fashion. Neither of these alternatives are very practical nor easy to use, so the intermediate files are simply raw memory dumps of a single array. The files are designed to minimize the overhead of dumping data to files, and during experiments the files are stored on a `tmpfs`, so the data is only kept in memory and is never written to disk[42].

The file format is literally raw column-major multidimensional array data, nothing more and nothing less, which makes it possible[10] to memory map the entire file and obliviously use this as the storage backend for any regular array. Metadata is stored as part of the file suffix[11], and is parsed/generated by the applications when the files are read/created.

4.1.3 Instrumentation

An early version of one of the benchmarks was designed to decouple the particular computation task from the remaining parts of the benchmarking framework. This was implemented by having one orchestrating process that spawned differently implemented workers. The idea was meant to avoid subtle differences in the chunking procedure, but it turned that the thread spawning process imposed such a considerable delay, that any timing measurements would have been useless. This approach was eventually scrapped in favor of a

9. Calibrated scene exported as BEAM-DIMAP, image data converted from network to host endian and pixels reordered from row-major to column-major.
10. Memory maps have the possibility of being offset, but only page-aligned[20, `mmap()`], and a page is a fairly large overhead for a dozen bytes of metadata.
11. `cmx` file suffix := `.cmx_`d_1`x`$\cdots$`x`d_n, `x` being an ArrayFire data type[45, `enum af_dtype`] (e.g. `f32` for single floating point) and d_i for the size of i-th dimension.

simpler chunking scheme reimplemented by each worker, but the lessons from this detour influenced future timing instrumentation.

The initial timing instrumentation was external[12], because the framework's bootstrapping time was considered a relevant part of the total benchmark execution time. This approach would automatically bias native applications, because others would also be spawning their accompanying runtime system, but this bias is natural, and not particularly wrong.

The idea behind taking bootstrapping time into account was based on the assumption that the applications were supposed to be standalone, as in, who uses an interactive Python session to start their programs? In a scenario where the execution time of running a Python script is compared with that of running a native applications the difference is noticeable, but the overhead is in the order of tens of microseconds. Then a GPU MATLAB implementation was developed, which was frustrating enough just to develop into a standalone application[13]. Running this implementation highlighted the fact that the spawning of the MATLAB interpreter itself takes about 10 seconds alone, and that sort of handicap (i.e. longer than many of the experiments) for simply being an interpreted language would have been a bit too much. It also turns out that lots of MATLAB developers do actually keep a running interactive programming session where they both develop and execute their code, who knew?

The result of this observation was a change of mind when it comes to timing measurements, so from that point onwards measurements were being taken, and reported, internally by each implementation. The distinction between bootstrapping and other kinds of initializations are a bit vague, so, when bootstrapping delay is already ignored, then the measurements are just as well taken as close to the actual benchmark as possible.

Event based timing data relative to the start of the applications are logged to the application's output, then the log output is being post-processed to convert timestamped events into total runtime. The timestamps are based on wall-clock, as opposed to CPU, time, so that potential multiprocessing within frameworks is counted as total user observable runtime, instead of thread-accumulated CPU resource consumption, because that would be a very irrelevant measure for comparing GPU applications.

12. At that point the total runtime was measured by wrapping the application execution with the shell's `time` measuring functionality.
13. Its "`main`" function ended up being wrapped by a bootstrapping shell script, in order for the result to be capable of shell execution.

4.2 Streaming Pipeline Benchmark

This benchmark is a SAR image data preprocessing step that extracts statistical features from polarized SAR data into a more suitable basis for subsequent terrain clustering.

The preprocessing consists of a noise-reduction step and a part that extracts a set of statistical features emphasizing terrain differences. Intensity SAR data is more or less regular pixelated image data where the different polarizations can be thought of, and are often visualized, as colors. The noise-reduction part is a blur filter[14] and the feature extraction relates to converting the image's color scheme (e.g. Red, Green and Blue (RGB) to Hue, Saturation and Value (HSV)) in order to extract other kinds of information.

All, but the averaging process, work on elements independently, and the averaging process is element-wise independent given read-only (i.e. separate from output) input data, so the entire process is more or less embarrassingly parallelizable. Additionally, the locality dependency of the averaging process is limited to a fixed window centered around every pixel, so the workload can be chunked into arbitrary submatrices given half a window of padding pixels around the edges of its input.

4.2.1 Multilook

SAR data has inherent speckle noise caused by how the illuminated surface scatters the reflected radar waves[15, 1.8]. A common noise reduction approach, and the one that is utilized in this benchmark, is known as multilook. Multilooking trades off resolution with increased signal-to-noise ratio by averaging away the uniformly distributed speckle noise. One of the methods for multilooking SAR data is by low-pass filtering the measurement intensity[15, 3.8], and the implementations in this project does that by averaging every pixel with the values of neighboring pixels within some given range.

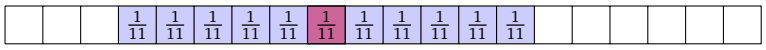


Figure 4.2: Convolution filter (overlapping blue) around the red pixel. Window size: $l_x = 11$, and constant weights: $\forall_i w_i = \frac{1}{l_x}$ (i.e. averaging)

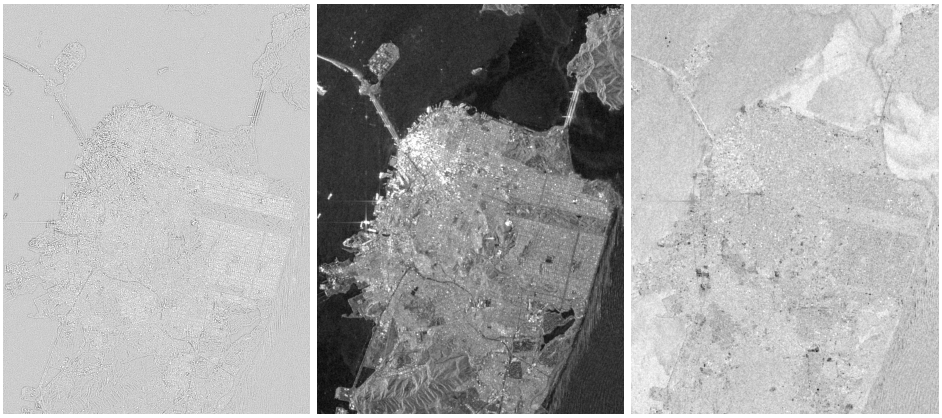
Figure 4.2 visualizes an 1-dimensional version of the filtering procedure for a given pixel. Convolution is a generic type of filter where relative neighboring pixels are associated with their own weight, and the resulting value for the

14. For multilooking every pixel is weighted equally, whilst regular blur filters use a Gaussian convolution (i.e.. weighting diminishes with radius from origin pixel).

origin pixel is the sum of these weighted pixels. In this particular usage case the filter is an equally weighted average (i.e. constant weights), so some of the implementations are optimized by scaling the sum of the raw pixels, instead of scaling each pixel before they are accumulated.

The procedure averages pixels along one dimension at a time, which has $\mathrm{O}\left(\sum_{d=1}^{n_{\text{dimension}}} l_d\right)$, with l_d being the window size (i.e. the number of neighboring pixels to consider) for a given dimension. Since the input depends on the previously averaged dimension, this approach is not perfectly parallelizable, but it easily outperforms the naive, and equivalent, approach of doing all dimension in a go, because that has $\mathrm{O}\left(\prod_{d=1}^{n_{\text{dimension}}} l_d\right)$. The computation needed for the averaging process scales linearly with the window size, so this is left as a parameter[15] to the benchmark.

4.2.2 Feature Extraction



(a) $\log \mathrm{RK}$ **(b)** $\log \mathrm{MRCS}$ **(c)** $\log \mathrm{R}_{cr}$

Figure 4.3: Extended probabilistic features computed in Section 4.2. Averaging window: $l_x = l_y = 5$. Cropped to the subset passed along to Section 4.3.

The intention behind this benchmark is to convert the image data into a basis that better represents the variation of the illuminated surfaces, and the result is visualized in Figure 4.3. The idea behind this visualization is to show that the different features are emphasizing distinct areas of the image, but urban areas are complicated scenes with lots of details, so this distinction is not that clearly visible.

This thesis focuses on the computational problem, so for precise definitions of,

15. For multilooking, the window size is realistic in the range 5 to 100 depending on what the SAR data is supposed to be used for

and more discussion about, the particular features, see [13, 2.2].

Relative Kurtosis (RK) is a property of how each pixel relates to its neighboring pixels, so its computation incorporates the same averaging procedure as the multilooking process. Apart from that, the entire feature computation boils down to $3n_{\text{polarization}} + 7$ elementwise math operations.

4.2.3 Chunking

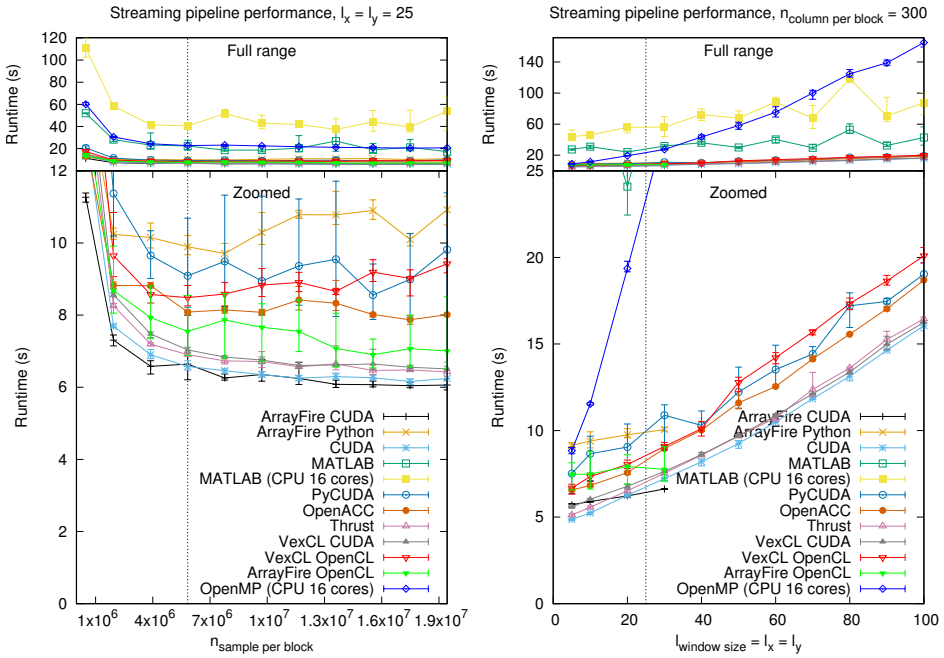
The workload is intentionally chosen to be larger than what can fit within the memory area available to the GPU, which requires the process to work on a subset of the workload at the time and stitch the result of every subset computation together into one coherent output. The chunking procedure has been designed to be as simple as possible, in order to avoid accidental differences between implementations, and therefore, minimize the extent to which the chunking part of this benchmark impacts the results.

Matrices in this project are stored in a column-majorly fashion, and, for simplicity and memory localization, the chunking process operates on sets of entire columns. These column subsets overlap the input data of bordering chunks with enough padding to accommodate enough neighboring pixels for the averaging processes to yield valid results for the entire chunk. In a real application the chunk size would be estimated based on the available GPU memory and the required memory footprint, in order to maximize the resource utilizations, but, for the sake of the experiment and to simplify doing it identically across multiple implementations, this is left as a statically configurable parameter of the benchmark.

4.2.4 Performance Results

Figure 4.4 presents the performance results from the streaming pipeline experiment. The different subfigures plots total runtime as a function of either the chunk size with a fixed multilook window size (i.e. workload) (Figure 4.4a); or the other way around (Figure 4.4b).

First of all these plots have two CPU versions: MATLAB (yellow) and OpenMP (blue diamond); and two OpenCL versions: ArrayFire (green) and VexCL (red). The OpenMP version have been added as a baseline because this is an embarrassingly parallelizable problem, so it required a total of 5 pragmas to a sequential C++ implementation that was needed by one of the frameworks anyways. High-level GPU MATLAB code is a matter of initializing arrays on the GPU instead of the CPU, so the MATLAB CPU version have simply been



(a) Varying block size. **(b)** Varying multilook window size.

Figure 4.4: Streaming pipeline performance. Lower is better. Error bars indicate min-/maximum runtime out of 5 repetitions. The dotted vertical lines indicate the fixed value for the other subfigure.

generated by replacing all of the GPU array initialization with initialization of regular arrays. The OpenCL versions are from frameworks that support interchangeable backends, so they are simply compiled with other parameters than their CUDA counterparts.

CUDA (cyan star) is the low-level GPU implementation, and a low-level sequential C++ implementation have been used as the identical basis for automatically parallelized OpenACC (maroon filled circle) and OpenMP implementations. Thrust (pink) and VexCL (gray) are different frameworks where the kernel code have been generated from abstract C++ templated code instead of being handwritten in CUDA C. ArrayFire (black), MATLAB (teal) and the PyCUDA (light blue circle) implementations are written as mathematical operations on matrix types.

Thrust and PyCUDA utilize the multilook procedure from the low-level CUDA implementation, but apart from that all the implementations are written utilizing their respective frameworks' API.

4.3 Compute-Intensive Clustering Benchmark

External memory transfer is expensive on GPUs, and the benchmark described in Section 4.2 primarily represents problems where such behavior is unavoidable. However, that is not always the case, and, whenever possible, GPU applications tend to minimize the need for external memory transfers. The following benchmark is a pattern clustering algorithm used to detect distinct terrain classes, and it avoids unnecessary memory transfers by limiting its memory footprint to something that fits within the available GPU memory.

The algorithm depends on frequent reductions over the entire workload, so having a workload larger than the available GPU memory instantly causes thrashing problems, and in such a scenario getting any notable performance would be unrealistic. Scaling up the GPU memory space by chunking the workload among a cluster of GPUs with enough combined memory, and only transferring partial aggregates between them, is a possible workaround to the thrashing problem. However, clusters of GPUs are outside the scope of this thesis, and, in any case, there already exists extensive research on the topic[9, 14].

4.3.1 Automatic Mixture of Gaussian

The clustering algorithm is known as Mixture of Gaussian (MOG)[13, 2.4], and, briefly described, it uses an Expectation-Maximization (EM) algorithm to iteratively converge a given set of statistical models[16] such that they best fit distinct subsets of samples. This algorithm considers samples independently and is agnostic to what constitutes a sample. For this particular problem the samples are vectors of features precomputed by the benchmark described in Section 4.2.

The automatic part of the algorithm refers to it starting with the entire dataset modeled as a single cluster, and hierarchically splitting the worst fitting cluster until every cluster fits some distinct set of samples fairly well, or it reaches some given maximum number of clusters. Determining the worst fitting cluster constitutes a fairly comprehensive weighted binning procedure, but refitting the clusters with the EM algorithm after each split is still the primary computational hotspot. When the clustering algorithm completes, the final set of statistical models, which can be used to classify samples, is returned (i.e. saved to filesystem) for future use (e.g. testing correctness).

16. Mean and covariance of the samples within that cluster and the overall probability that a sample belongs to that particular cluster.

The number of distinguishable clusters depends on the particular set of samples, but the overall trend is that the number of clusters the algorithm considers well-fitting rises with the amount of available samples, and during the experiments it ranges from a couple to a maximum of 20 clusters. Predicting how many EM iterations are needed to refit a particular set of clusters is no easy task, but the trend seems to be that it ranges from tens to a couple of hundred iterations.

Each EM iteration requires many reductions over the entire set of samples and lots of elementwise operations, so the amount of computation per sample during a single iteration amounts to a fair bit more than that needed for a single sample in the data pipeline benchmark. And, when these iterations are repeated hundreds of times, the total amount of computation easily makes copying the input samples to the GPU and the final statistical models from it neglectable with respect to the total performance.

4.3.2 Data Reduction

This is a computational expensive algorithm, and its computational effort is more or less proportional to the amount of input data it is given, so giving it tremendous amounts of data causes a tremendous amount of work. The cropped satellite image visualized in Figure 4.1b already contains enough samples to start making problems, and that image covers a fairly limited geographical area. Reducing the amount of computation is useful, if not essential, when it comes to getting results in a timely fashion (e.g. running my benchmarks), but that is not the only downside of working with large datasets. One of the insolvable problems is that the cuBLAS[30] library, which is an internal dependency in some of the relevant frameworks, has a per dimension limit[17] of around a million samples. Another more practical, and even more limiting, problem is that the part of the algorithm that determines what is a fairly well fitting set of clusters works best with a fairly limited number of samples.

This particular project focuses on the computational load, so it is irrelevant to the experiments themselves which part of the data the benchmark ends up classifying. However, in any practical application the clustering task would be associated with a specific area to work on, so simply taking the n first samples, or some other arbitrary submatrix for that matter, would not necessarily represent the intended area.

17. This library seems to be limited to 2-pass reductions, and each pass reduces elements blockwise, so there is a upper bound of $n^2_{\text{Threads per block}}$ elements, which is 1024^2 on current hardware.

The resulting classifiers' application domain is the type of data that the clustering algorithm was given, so, in order for them to classify a specific satellite scene, the clustering algorithm must be given samples that represents the terrain in that particular scene. Regular image downscaling uses filters that yields pixels striving to represent the diversity of what was removed (e.g. blockwise averages). That sort of data reduction scheme is an example of a method that would not preserve the properties of original samples, and which would yield a set of classifiers not suitable for the original dataset. It could, for instance, include a cluster that represents the average of a forest, whilst in the original scene that particular forest could consist of separate forest patches with varying degree of denseness, all of which differs from their combined average.

The data reduction approach in this benchmark is linear subsampling (i.e. dropping all but every *n*th sample). This approach uses real samples, and hopefully, gives a statistical representative view of the diversity within the intended area. The subsampling rate is used as a parameter of the benchmark's computational workload, but it should be mentioned that different subsampling rates yields entirely separate views of the data, so there is no guarantee that they yield the same clusters or need the same amount of work to complete.

4.3.3 Performance Results

The clustering workload is not directly proportional to the number of samples, so any interpretation of the following results requires some background information about which factors impact the application's workload. The workload is somewhat dependent on the size of the input data, because mathematical operations scattered around the benchmark work with vectors and/or matrices that are proportional to the number of samples, so there is an underlying trend that the amount of work is proportional to the number of samples.

However, the total workload depends on how many iterations the MOG algorithm needs before it considers each set of cluster to be converged, and how many clusters the application deems necessary for them to fit the data well enough. Section 5.2.4 concludes that this benchmark's major computational hotspot is the process that converges the statistical models, and each iteration of this process requires the same amount of work, but this amount depends on the number of clusters the process works on at that particular point in time.

These benchmarks are supposed to be deterministic, so, although the change in workload is not proportional to the change in input size, that does not mean that implementations are not comparable when they are given identical input. However, there is a caveat that the algorithms aggregates lots of single precision

floating point values, which also aggregates their individual inaccuracies, and both the convergence procedure and the test to determine whether a set of clusters is well fitting enough (i.e. the application's exit condition) are very sensitive to inaccuracies. GPU hardware has support[18] for IEEE standardized floating point calculation[31, 4.4], but there may be subtle differences between the implementations, and, in any case, they are not necessarily ordering their floating point calculations identically. The end result is that there are considerable workload fluctuations among the different implementations, which need to be taken into account when the performance data is analyzed.

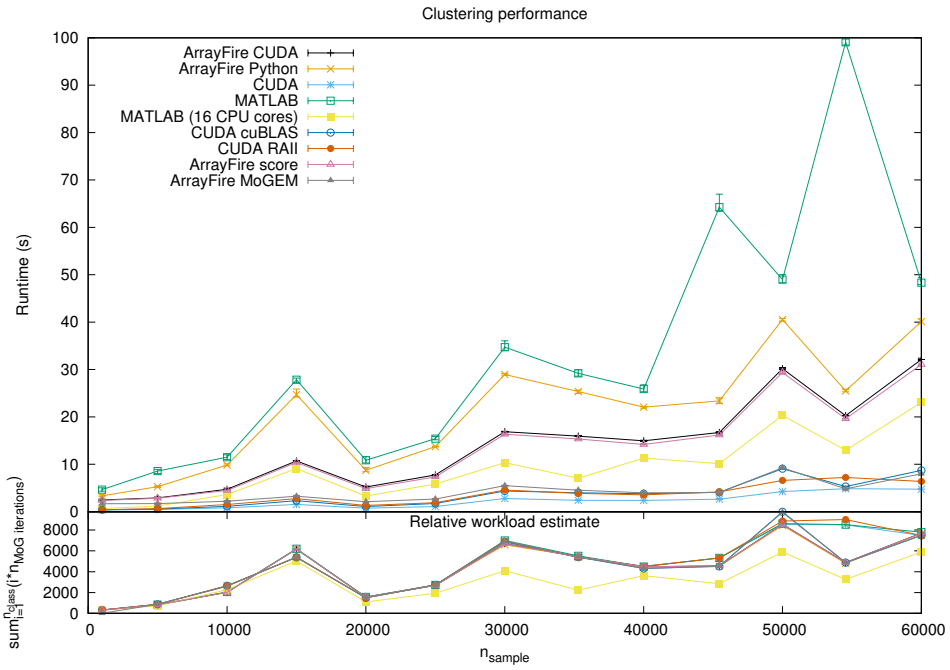


Figure 4.5: Clustering performance with varying input size. Lower is better. Error bars indicate min-/maximum runtime out of 5 repetitions.

Figure 4.5 presents the performance results from the clustering experiment. These results have been plotted alongside an estimate for how much workload each particular implementation went through during the measurement, because any relative performance will need to consider their respective workload.

In order to produce more easily comparable results, Figure 4.6 presents results from the same benchmark, but in this experiment the workload has been

18. They also support a faster alternative, and whether to use one or the other is software configurable, so there is always the possibility that some of the frameworks have disabled IEEE standardized calculation internally.

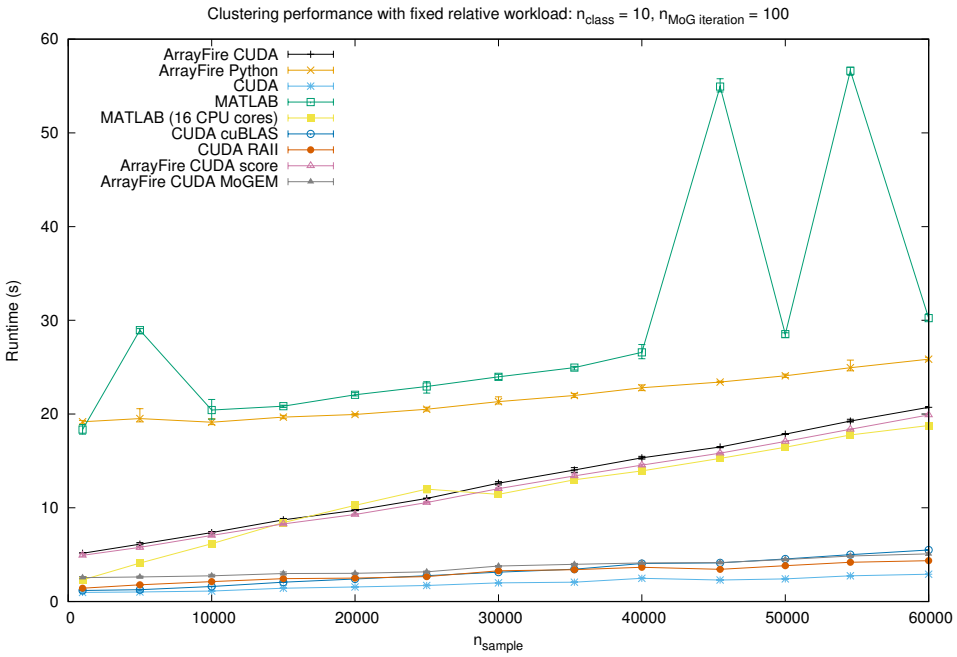


Figure 4.6: Clustering performance with varying input size, and parameters forcing workload to be proportional to the input size. Lower is better. Error bars indicate min-/maximum runtime out of 5 repetitions.

forced to be proportional to the number of samples. The difference between this simulation and the real application is that the amount of work needed to consider a set of cluster models converged and when clusters are considered well fitting are fixed, instead of being determined by the application.

These experiments consist of 4 base implementations: ArrayFire (black), Python bindings to ArrayFire (orange), MATLAB (teal) and low-level CUDA (cyan star). As with the previous experiment, the MATLAB version also has a CPU derivate (yellow), but the ArrayFire OpenCL version have been dropped in favor of the possibility of substituting parts of the ArrayFire implementation with parts from the low-level implementation. There are also some minorly diverging versions of the CUDA implementation and the ArrayFire implementation.

Development of a PyCUDA version was also started, but that development track was eventually abandoned. Nevertheless, that work lead to the development of low-level customized matrix product functionality, which, along with the reasoning for dropping the PyCUDA implementation, are discussed further in Section 5.4.1. This functionality is used by the basic CUDA implementation, but it also yielded a diverging implementation, CUDA cuBLAS (dark blue), which instead uses equivalent cuBLAS routines for calculating matrix products.

There was also some work on implementing a Thrust version, but it turns out that trying to implement the benchmark using vector iterators would require way too much effort than it could ever be worth, so, as with the PyCUDA implementation, this development track was eventually scrapped. Thrust is designed to cooperate with CUDA, so there is not necessarily anything wrong with delegating that which is easier to implement as low-level code to CUDA, and the results is the CUDA Resource Acquisition Is Initialization (RAII)[19] implementation (dark orange). This diverging version utilizes the Thrust vectors as dumb data containers, whose allocated memory is managed as a regular C++ object (i.e. allocated when it comes into scope, and deallocated when it leaves the scope), as opposed to the optimized CUDA version, which allocates all the memory it will require during its entire lifetime before it starts doing any computation.

The ArrayFire forks are versions where parts of the algorithm have been substituted with the same low-level implementations that is used in optimized CUDA versions. These substitutions are allocating temporary arrays, so their results are most comparable to the RAII version of the CUDA implementations. The score version (pink) optimizes the scoring procedure and the MoG Expectation-Maximization (MOGEM) version (gray) optimizes the convergence procedure.

19. Named after the C++ resource management scheme.

5 Discussion

This discussion starts in Section 5.1 with a summary of some notable differences between the separate implementations, which will become relevant when these frameworks are being compared. Then it continues with the performance evaluation in Section 5.2, before you all have forgotten the results from the experiments. In Section 5.3 it continues with briefly describing how the correctness of these implementations has been tested, and this chapter finishes with Section 5.4, which is an evaluation of my experiences from programming using these different frameworks.

5.1 Notable Implementation Differences

The loop structures in the basis for the OpenMP implementation is optimized for GPU and other forms of highly parallelized implementations, while optimized multiprocessing CPU implementations have a whole other set of considerations when it comes to how a problem is optimized. This implementation should therefore be considered more of a proof-of-concept than an optimized CPU implementation. The averaging task at hand, for instance, has much better solutions if you work on an entire window of data the same time, which is doable with static workload orchestration of continuous data blocks, which is the approach used by naive parallel-for constructs in OpenMP.

The ArrayFire implementations, among others, utilize convolution function-

ality from their library, as an alternative to the averaging procedure in the pipeline benchmark, and ArrayFire's particular convolution implementation is limited to a window size in the interval $[0, 31]$, which is the reason why these implementations drops off after the 30 measurement in Figure 4.4b.

The streaming pipeline implementations are designed to minimize the amount of unnecessary data buffering, but the extent to which this is possible depends on the framework. All the C++ implementations work on memory mapped buffers provided by some common chunking module, so these applications have no CPU buffers. Input/Output (I/O)-backed memory mapping only works for valid file data (i.e. areas beyond the End-Of-File (EOF) are not writable), so, before memory mapping of the data, the C++ implementations seek to where the file should end, and writes a zero, which implicitly allocates zeros between the previous EOF (i.e. start of file) and the newly written byte[20, `lseek()`]. MATLAB supports memory mapping, but it lacks the support to seek beyond the EOF, so it is limited to explicitly writing chunk after chunk to output files, which requires it to have one, or more likely two[1], application buffers of the data. Python also have memory mapping support, but the Python bindings for ArrayFire do not support dumping an array directly to a memory address, so the outputs are application buffered as regular NumPy arrays before they are copied to the memory mappings.

Some statistical functions used in the scoring part of the clustering algorithm were missing from all of the GPGPU frameworks, so the calculation of those are delegated to the CPU. This imposes some unfortunate memory transfer of the arguments and the results, but it concerns fairly small amounts of data, so it is acceptable, and, in any case, it presents a scenario that is relevant for discussion. More about that in Section 5.4.1.

All the OpenCL backends, the VexCL CUDA, and the PyCUDA implementations are JIT-compiling the GPU kernels. They cache their compiled kernels for the rest of the application's lifetime, so the compilation only yields an initial compilation overhead, and VexCL even caches its kernels persistently (i.e. writes them in the user's home directory) for later runs. This gives these implementations the possibility of optimizing for known runtime parameters (e.g. fixing thing to size of input and better loop unrolling heuristics), but the fact that the compilation process impacts runtime performance also discourages computationally expensive optimization. The CUDA compiler, on the other hand, is designed to perform very extensive optimizations to the GPU code during its compilation phase.

1. One copy at the user-level and one in the underlying C file handler.

5.2 Performance

First of all, the CPU and OpenCL versions have been added to help put the results in perspective, but these will not be the focus of the discussion. And, when we are on the topic of CPU implementations, the backend used for OpenACC/OpenMP could also have been compiled as a sequential version, and there is a sequential ArrayFire CPU backend, but both of these have been left out of the experiment because they are an order of magnitude slower then the already slow OpenMP version, so they would have skewed the results, and increased the time it would take to run the benchmarks, which are already taking an annoyingly high amount of time to finish.

5.2.1 Runtime Variation

The values plotted in Figure 4.4, Figure 4.5 and Figure 4.6 are means over 5 repeated measurements, with error bars indicating the minimum and maximum runtime measurements, as a means of indicating that these are repeatable results. Each experiment (i.e. running every repetition of all implementations with one fixed set of parameters) randomizes their execution order, as a means of distributing the measured impacts from potential error sources evenly among the implementations.

This is a dedicated machine, and there is no intentional background load whilst running experiments, but the machine runs regular Linux with a graphical user interface, so there is always lots of background processes that can have minor impacts on the measurements. Figure 4.4 shows that the runtime of the multiprocessing OpenMP implementation varies most of all them all, but this is to be expected, because that implementation utilizes all the CPU resources it has available, which makes its performance much more susceptible to the load variance of the other background processes. The multiprocessing CPU version of the MATLAB implementation measures much more stably than the OpenMP implementation, but in comparison the MATLAB implementation does not, maybe even intentionally, utilize more than about half the available CPU resources, so then there is lots of spare load left for the other background processes.

In all the experiments the trend seems to be that the measurements from GPU implementations in interpreted languages, MATLAB and Python in this case, seem to vary a bit more than the implementations in compiled languages, but, that being said, there are also noticeable variations among some of the measurements for implementations in compiled languages. However, the overall measurement variance is not very substantial, and, in any case, whenever measurements from separate implementations are periodically bordering each

other's variance they are considered having more or less equivalent performance.

5.2.2 GPU vs. CPU

The streaming pipeline experiment shows in Figure 4.4b that given enough work on the GPU, every implementation managed to outperform the multiprocessing CPU implementations. As pointed out in Section 5.1, the OpenMP implementation is not a fair opponent, but nevertheless the extent of the performance gap between it and the GPU implementations, and the fact that the CPU MATLAB version shows similar results, makes this a fair suggestion.

Even at a fairly small workload, most of them manage to outperform both 16-core multiprocessing implementations, despite needing to transfer data back and forth to the GPU. It is also notable that Figure 4.4a indicates that at smaller block sizes, and this is still in the order of megabytes, the overhead of spawning multiprocessing threads (i.e. OpenMP) outweighs the overhead of transferring data and executing parallel code on the GPU. Although not a very novel observation, it was still a surprising results, given the intuition that GPU memory transfers are fairly expensive operations.

The Figure 4.6 does not show as clear a distinction between GPU and CPU implementations for the clustering benchmark, which might indicate that some implementations would prefer larger computational task. The CPU contestant in this benchmark is a MATLAB implementation, and, although I am neither postulating that I write perfect MATLAB code nor that there are not any better suited frameworks available, this algorithm is smack in the middle of MATLAB's intended problem domain, so the various matrix operations should be fairly optimized.

The fact that the GPU MATLAB version is consistently worse than its CPU counterpart indicates that the extent of the code MATLAB manages to parallelize to the GPU is outweighed by the work it requires MATLAB to do on the CPU.

5.2.3 GPU vs. GPU

There was supposed to be little focus on OpenCL, so let us start by getting those implementations out of the way. ArrayFire and VexCL were the only implementations with OpenCL backends, and these backends were only a part of the streaming pipeline experiment. The ArrayFire version beats the VexCL version with about a second in total runtime, and the ArrayFire version

is consistently in excess of a second slower than its CUDA alternative. These results are not very surprising, given the fact that these CUDA version uses precompiled[2] kernels, whilst OpenCL implementations compiles them as part of the applications, which easily can add a second to its total runtime, and, in any case, these differences have not been explored any further.

Streaming Pipeline

Both Figure 4.4a and Figure 4.4b places the MATLAB implementation about 15 seconds above any other implementation, which for these parameters constitutes about a doubling in runtime, but at least it does not seem to be increasing any more than the others when the workload increases. The caveat described in Section 5.1 about some double buffering of the output, might be one of the relevant factors, but, as the other implementations suffering similar problems fall much closer to the remaining implementations, this alone would not be enough to explain this performance difference.

The other implementations in interpreted languages (PyCUDA and ArrayFire Python) seem to have more or less equivalent performance, and they seem to be the next worse implementations with a runtime trending at between 3 and 5 seconds longer than the best implementation. After that, the OpenACC implementation, with in excess of a second longer runtime than the best implementation, is the only remaining implementation whose performance is separable from that of the remaining implementations.

Interpreted languages are usually a bit slower than native ones, so the fact that those have a bit higher overhead is not that surprising. The runtime differences between the frameworks have intentionally been stated in absolute terms, because Figure 4.4b clearly shows that an increase in workload causes a parallel shift to the runtime of all the implementations, which indicates that the differences between the frameworks is a more or less constant overhead.

If these runtime differences had been anything but constant, then there might be a reason to explore optimization paths further, but you would need some very specific reason for justifying any notable development effort resulting in shaving a couple of seconds from the total runtime of any application. That brings us to major highlight from a benchmark like this one, which is that if you are computing fairly little on a GPU in one go, then the room for potential optimization and differences between high-level and low-level frameworks will be fairly limited. The results from the MATLAB implementation shows that there are frameworks that internally clearly do something wrong, and analysis

2. VexCL JIT-compiles CUDA kernels also, but these are cached from previous runs.

shows that in this particular case the problem is memory access. More on that in Section 5.2.5.

Clustering

Once again, Figure 4.6 presents MATLAB with the price for longest runtime, but this time there is a much greater span amongst the different frameworks. The other implementations in interpreted languages, which is now, unfortunately, down to only the Python binding for ArrayFire, grab the next worst position, but that is understandable, considering their remaining opponents is the equivalent C++ implementation and even more optimized ones. However, this time ArrayFire separates itself much more clearly from the low-level implementation, so, given this more computational expensive benchmark, it is clear that this time, implementing code in a low-level framework has a much greater impact on the overall performance. MATLAB suffers similar problems as before, and all hope is not lost for ArrayFire, but more about that in Section 5.2.4.

The most notable observation from Figure 4.6 is that workload increase yields a much steeper increase in the C++ version of ArrayFire than its Python binding counterpart. This can probably be associated with the CPU parts of the Python implementation requiring more time to execute, which results in more GPU idling, so increasing workload probably fills up some buffer of underutilized GPU resources, which causes a smaller impact on its total performance. However, and as the trending at the end of Figure 4.6 begins to indicate, these implementations will surely level off eventually, because having the Python bindings surpassing the performance of the equivalent native C++ application would be a fairly unrealistic scenario.

5.2.4 Hotspot Exploration

All of the GPU frameworks that are a part of this survey have some sort of method for incorporating low-level GPU code. This facilitates scenarios where you start by writing the entire application in a high-level framework, then you figure out which parts are worth optimizing, and only reimplement those, as opposed to the entire application, as low-level GPU code. Such an approach is a compromise between development effort and the performance of the applications, and prioritizing the optimization of code snippets that have the most impact on the performance have a tendency to pay off in the end.

Instrumentation

There is no fundamental difference in how you profile execution time for GPU applications in comparison to CPU applications, but there are some pitfalls. First of all, many profilers default to measuring CPU time consumption, which yields irrelevant, or even misleading, information when instrumenting the GPU parts of an applications. This is caused by the GPU working in parallel with the CPU, and, if you are not explicitly using the CPU for other computation, it usually blocks idly on some sort of GPU synchronization primitive[3], which usually does not consume any notable CPU resources.

The profilers that measure wall-clock time spent at the different points in the program do not have the same problem, but they are still tracking where the CPU is spending its time (i.e. at the synchronization points). However, if there are synchronization points (e.g. returning some value computed on the GPU) in a function, then the computation time of that function is accumulated correctly at the granularity of said function, but it is important to remember that time spent in preceding functions without synchronization points might also be accumulated at this point. This works well for APIs that synchronize everywhere, and at a granularity where the CPU somehow has to be involved with planning of the GPU's execution flow, but it requires intimate knowledge about where the code synchronizes the CPU and the GPU.

There are tools, such as nvprof[33, 3] for CUDA, that profiles GPU execution time, but their results are not always as easy to backreference with the CPU application code. If you have specialized kernels that are only called from one place in your code, then this is very helpful to determine where the computational hotspots are in your code, and, of course, you are given information about which parts of the GPU code would most benefit optimization. General-purpose frameworks, on the other hand, either use general-purpose kernels (e.g. ArrayFire's `reduce_first_kernel`), which will be scattered around the source code and nevertheless aggregated together; or generate ones with undecipherable[4] names. The general purpose ones tend to have fairly descriptive names (e.g. matrix multiplication kernels usually includes the BLAS abbreviation *gemm*), so you can get a feeling of which types of code is relevant for optimizations, but, other than that, the GPU profiling data is not very useful to these general-purpose frameworks.

3. [32, `cudaDeviceSynchronize()`] will explicitly wait for the GPU to finish what it is doing, but most commonly applications block when waiting for results (i.e. [32, `cudaMemcpy()`]).
4. I have come across generated kernels aggregated under a single name, as with `vexcl_vector_kernel`; arbitrarily named ones (e.g. `KER6817788380830779747`, one of the JIT-compiled ArrayFire kernels), and ones that unambiguously describe their code, such as Thrust kernels named after their expanded recursive C++ template, of which the smallest I have seen is a constant initializer whose name is in excess of 800 bytes long.

Nvprof has a mode that periodically samples the CPU call stack, which turns it into a statistical version of one of those wall-clock time CPU profilers, but unfortunately this data is completely uncorrelated to the GPU profiling data. Nevertheless, this mode is what is used in the following analysis for the C++ implementations, but this tool only profiles native function calls, which is not easily translated into statements in interpreted languages, so this is more or less useless for those implementations. Instead, those implementations use internal profiling, where the MATLAB versions uses [28, profile] and the Python version uses the cProfile[41, 27.4.3] module, but I have not checked whether these use CPU or wall-clock time for their internal measurements.

Hotspots in Clustering Implementation

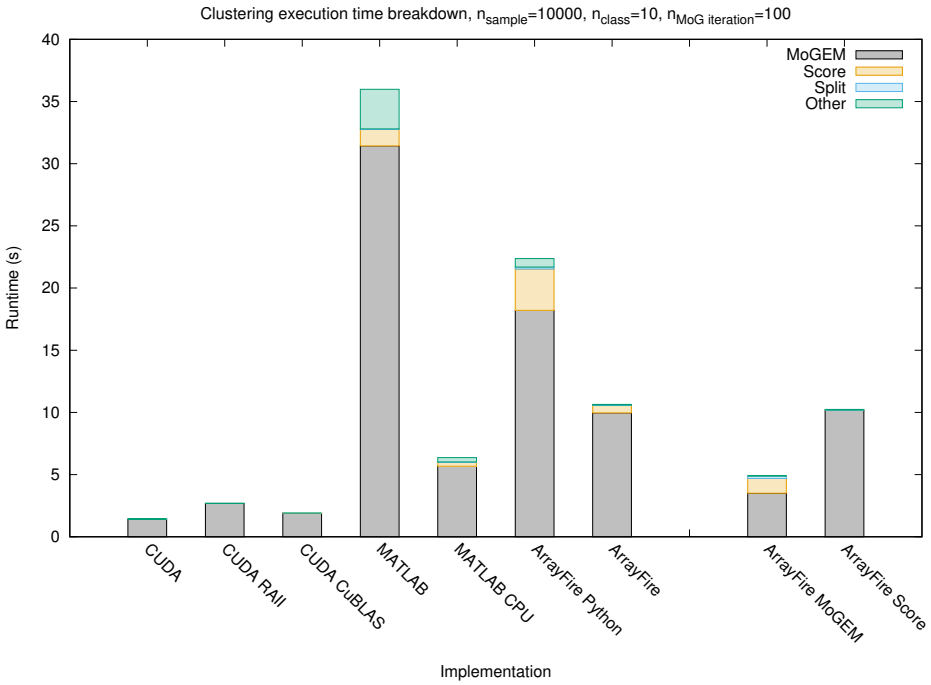


Figure 5.1: Runtime breakdown of the clustering implementations. Those to the right are the optimized versions.

Figure 5.1 visualizes a very coarse-grained runtime breakdown of the clustering implementations, and it clearly shows that the MOGEM (the function that converges a set of cluster models) is the major computational hotspot in all of the implementations. The next most influential part of the program is the scoring procedure, but anything beyond that is more or less neglectable. The other category includes calls at the top-level to some functionality that is shared with the MOGEM procedure, which is the reason why this category

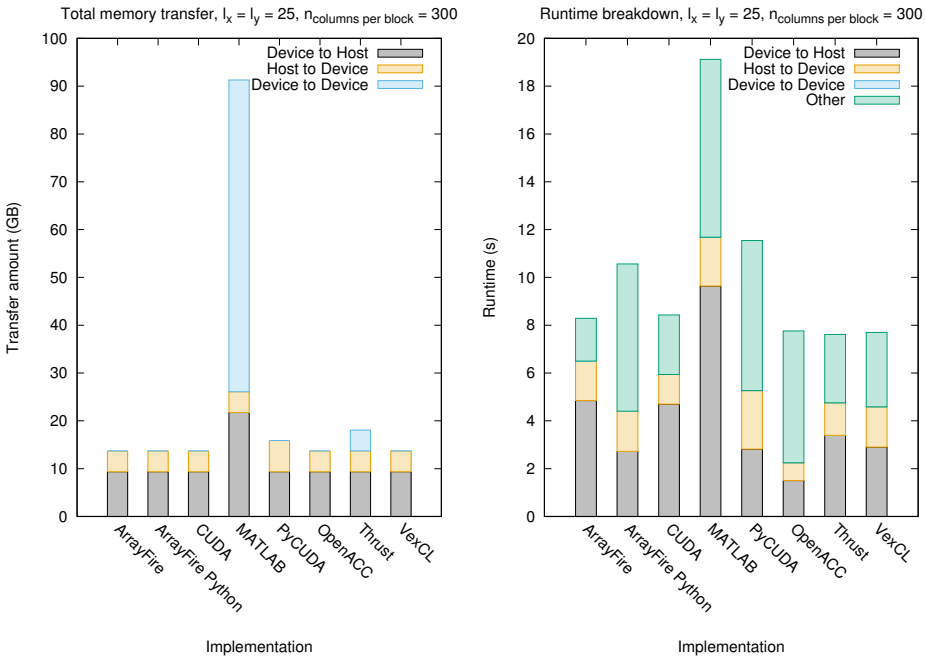
is disproportional in the MOGEM optimized ArrayFire implementation when compared to the original one. In the GPU MATLAB versions this category also includes a mean over the original samples, which is disproportional large in comparison with similar operations, so, as it is the first operation on the input array, it probably indicates that it includes the time of a lazy copy operation to the GPU, but, that being said, the timing scope in the performance benchmarks has a bit wider scope, so rest assured the equivalent operation in the other implementations is included in those measurements.

The implementations to the right in Figure 5.1 are the ArrayFire versions that have had some parts of them replaced with equivalent low-level implementations, and it is important to emphasize that they are also allocating temporary arrays, so their base of potential speedup should be the RAII version, not the fully optimized CUDA version. Both Figure 5.1 and the performance experiments indicate clearly that an optimization of the MOGEM procedure would have a major impact on that implementation's total performance, and Figure 4.6 indicates that it would have been even closer to the low-level implementations if this profiling session had used more samples. Optimizing the second most prominent hotspot has a much smaller impact, which is to be expected.

5.2.5 Memory Transfer

Figure 5.2 and Figure 5.3 show, respectively for the streaming pipeline and the clustering experiment, the amount of memory transfer and runtime contribution of memory transfers as recorded by `nvprof`. It should be noted that there are many factors impacting the achievable memory transfer speed, so this will vary a lot, and therefore, you should not look too much into minor differences in runtime breakdown. Additionally, the GPU is capable of overlapping memory and computation operations, and exactly how `nvprof` aggregates the different data in those scenarios has not been explored, but, if its graphical interface is any indication on how the raw data is sampled, then these measurements might overlap computational work.

If we ignore the MATLAB results for a minute, an rest assured those will be discussed shortly, these plots showthe exact reason why this project focuses on two separate benchmarks. Figure 5.2a shows that the streaming pipeline benchmark inherently requires lots of memory transfers, and the delay of these memory transfers make up a considerable part of the total runtime. This part of the total runtime forms a lower limit of the achievable runtime, and, as shown in Figure 5.2b, this constitutes more than half the total runtime in most implementations. Whatever optimizations you do to the computation part of an application will always be limited by the delay needed for necessary memory transfers, so if you have external memory intensive applications, like



(a) Transfer amount. **(b)** Runtime breakdown.

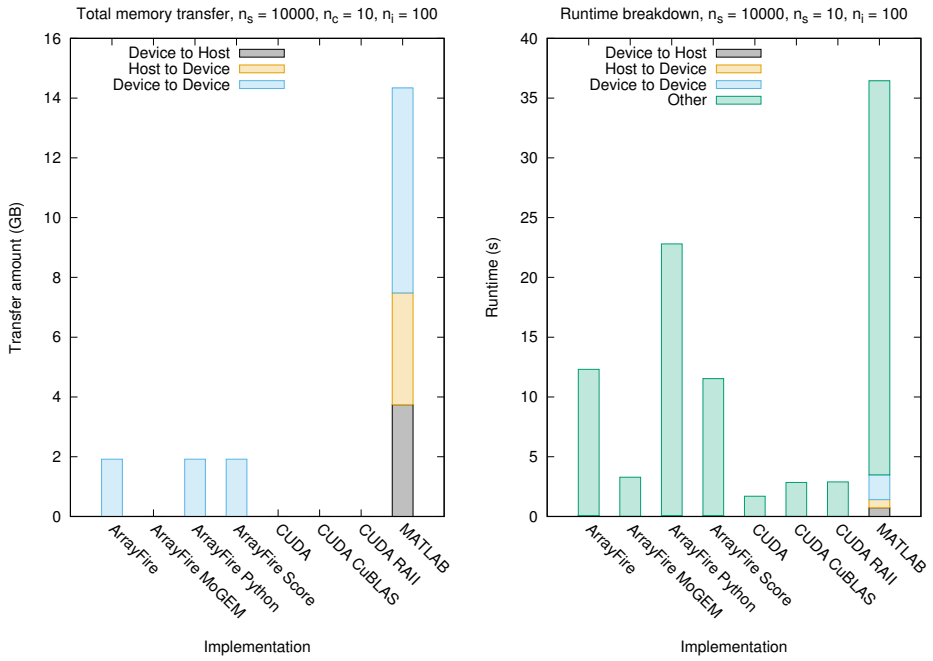
Figure 5.2: Streaming pipeline memory profiling. Lower is better. This is based on `nvprof` data (i.e. CUDA), so CPU and OpenCL are left out.

the streaming pipeline, then there are limits to the achievable gain of the effort put into optimizing that application.

Figure 5.3a shows the opposite scenario, where the need for external memory is neglectable, so, as shown in Figure 5.3b, its impact on the total runtime is also neglectable. Applications like this one do not have the same inherent limit to the potential impact of optimization, and, as such, the computational differences of the underlying frameworks are much more relevant to the performance of this application.

MATLAB

Now, let us come back to the discussion of MATLAB's behavior. In both of these benchmarks it clearly moves around a much greater amount of data internally on the GPU than any other implementation. The impact of moving data around internally depends on how the GPU handles these operations, and, during the streaming pipeline experiment, the average internal transfer rate would be about 8000GB/s, which is more than an order of magnitude higher than the GPU's theoretical memory rate, so the only explanation is that the GPU simply



(a) Transfer amount. **(b)** Runtime breakdown.

Figure 5.3: Clustering memory profiling. Lower is better.

claims to have moved the data. My best guess is that these memory operations results in either pointer swapping or something like copy-on-write access to the original memory internally, but this has not been explored further. In the clustering experiment, the impact of the internal data transfer is much more apparent, so in that case, the GPU is probably making actual copies.

The amount of externally transferred data, at least the data transferred to the GPU, is also substantially more than that of the other applications. The leading hypothesis is that the data arrays are proxies to their original data, so whenever they are actually used the data is copied from the original, which would be a terrible approach when data arrays span distinct memory spaces. The initial GPU array is used twice per polarization in the implementation, which combined with observations in Figure 5.2a that shows the MATLAB implementation transferring more or less twice the amount of data to the GPU than the others, seems to substantiate the hypothesis, but this has not been explored any further.

5.3 Implementation Correctness

This section concerns itself with how the implementations have been tested to ensure they are implementing their intended design correctly, and, as such, have comparable performance, so for a discussion on the correctness of the experimental performance results, see Section 5.2.1 and Section 4.3.3 instead. The focus of this thesis is on the benchmarks as programming and computational tasks rather than the potential real-world usefulness of the resulting implementations, so making them correct for the correctness sake was never a high priority.

The implementations have nevertheless been tested, and, at least to my knowledge, they are doing more or less the same as their reference implementations. Making them behave correctly is primarily a means of ensuring that each implementation is doing an equivalently sized task, and, as such, there have been little effort has been spent on making any of these implementations into any more than what is needed as a proof-of-concept for their respective benchmarks.

The streaming pipeline implementations have special parameters that are used to make them write their results to the filesystem, as opposed to some temporary memory location, which is their default behavior during performance testing. The correctness testing procedure involves visual inspection of results, such as those found in Figure 4.3, and a comparison with those generated by the reference implementation.

The clustering results (i.e. image classifiers) have been tested on real data, and the concept works, but neither the implementations created in this project, nor the reference implementation is doing a terribly good job with their given testing data, but, as their results are more or less the same, this should have minimal impact on their relative performance, and, as such, improving their accuracy was never prioritized. As discussed in Section 4.3.3, floating point inaccuracies tend to yield slightly different classifiers from the separate implementations, so a straightforward comparison of the results is out of the question, and the simple splitting of one more cluster tends to shift the color schemes in the visualization software that was used, which made visual comparison a cumbersome process. Instead a less comprehensive testing procedure that is based on specific splitting decisions is used. This procedure compares the cluster scoring at the split decisions and deems the implementations equivalent if they are making identical splitting decisions up to the point where one of them continues splitting more clusters than the other.

5.4 Programming Experience

This project reimplements real benchmark problems that were in no way prepared for being implemented as GPU software, and the choice of problems was intentionally outsourced to other parties. Outsourcing the choice of problems avoids having my inherent bias choosing problems that are, or for that matter are not, specifically well suited for GPU implementations. Implementing real problems also makes it possible that someone might get something useful out of the implementations themselves, which helps with my motivation.

Having specific problems means that the implementation obstacles that occur are neither fabricated, nor can they simply be avoided by redefining the initial problem. This section highlights some of these obstacles, and discusses how they influence the choice in, and the usefulness of, the GPU frameworks.

5.4.1 Feature Diversity

These problems are not the most complex ones in the world, but they require some mathematical features that are commonly delegated to library functions available to the programmer. The reference implementations were written in MATLAB, and this sections presents those of MATLAB's library features that needed custom implementation in some of the frameworks. The general idea behind this section is to point out that not all features are available in all of frameworks, so any choice of framework must be weighted against the potential need for implementing missing functionality that is required by your given problem.

Convolution

The reference streaming pipeline implementation used MATLAB's convolution[5] functionality to implement the averaging procedure described in Section 4.2.1. Library functionality for this was only available in MATLAB and ArrayFire, so the other frameworks needed custom implementations for this particular routine. Implementing this as a low-level kernel is a trivial task, and it opened up for the possibility of optimizing it by replacing the filter array with a compile-time constant, but, whenever possible, the same routine was also implemented in the high-level frameworks that was missing it.

The VexCL framework provides a `VEX_FUNCTION` construct for creating cus-

5. Actually it was using [28, `imfilter()`], but that is simply transposing its filter before convolving, and in this scenario, as the filter is constant, the distinction is irrelevant.

tomized functions, which expects a code block from the user and in it provides access to the original data pointers and an element index into them, but this is literally[6] just a feature that allows the user to write low-level code, so it is more or less the same implementation as the low-level one. The OpenACC version optimizes this sort of block automatically from plain C/C++ code (i.e. no external library support), so its implementation of this procedure might as well also be considered a low-level one. PyCUDA and Thrust supports customized elements-wise mapping procedures in their high-level interface, but that does not provide access to neighboring elements, so it is not enough, and they ended up calling the low-level routine.

The generality of Thrust's iterators makes this sort of problem solvable in its high-level interface, and development using those was explored superficially. The idea was to define these operations as the adding of the data vector with itself offset by the distance to each of the relative neighbors within the averaging window, but this approach escalated very quickly. It would have been possible to formulate this sort of operation as a set of recursive templates, but, frankly, this would be order of magnitudes more work than to implement it as plain CUDA code, so this approach was eventually scrapped in favor of calling the low-level routine instead.

Matrix Operations

The clustering algorithm works with matrices, and, in addition to element-wise operations, it requires matrix products and solving of linear equation systems. This sort of functionality is provided by the cuBLAS library, and the plan was to use this, or framework specific equivalent functionality, in the implementations. Out of the explored frameworks, the only ones that have native support for these operations are MATLAB and ArrayFire, and in these frameworks these native interface are used.

The low-level CUDA implementation intended to use cuBLAS for these operations, because the idea behind cuBLAS is that developers at NVIDIA optimizes these routines to the best of their knowledge, so that regular developers should not have to do that by themselves. The intention was that the other implementations also should either use cuBLAS directly or some native wrappers to it. However, it turns out that PyCUDA is not compatible with cuBLAS for technical reasons[25, FAQ:3.3], so the development of customized low-level routines for these operations was started as part of the development process for a PyCUDA clustering implementation.

6. This construct stringifies its given code block compile-time (i.e. not even parsing it), and runtime JIT-compiles this code either as CUDA or OpenCL code depending on its backend.

These routines were initially developed in the low-level CUDA implementation, because having them decoupled form the ongoing PyCUDA implementation made it easier to test their correctness during the development phase. A rudimentary low-level 2-pass routine for the kind of matrix products needed by this benchmark was developed, and used seamlessly by the partially developed PyCUDA implementation. Development of the linear equation solving functionality, on the other hand, turned out to be a more difficult tasks. Problems with large enough floating point rounding errors to throw the entire clustering algorithm off track was encountered fairly early in its development phase, which indicated that this part of the algorithm was so sensitive to such errors that getting this implemented correctly would require more work than it was worth. There was much effort into trying to fix those problems, because abandoning them meant scrapping the entire PyCUDA implementation and potentially others that turned out to be incompatible with cuBLAS, but eventually it was dropped, which also resulted in the PyCUDA implementation being scrapped.

Already having implemented the working matrix product functionality meant that this development track would at least yield something useful. The plan was to use this more or less naive implementation to point out that trying to achieve the same performance as the cuBLAS routine would require lots of optimization. However, as shown by, for instance, Figure 4.6, it turned out that this customized matrix product outperforms the cuBLAS version. The most prominent theory for explaining those results is that the customized routine is optimized for working on very skewed[7] matrices, whilst the cuBLAS routines tries to optimize for all intended usage cases. Other theories include only the custom implementation utilizing warp shuffling[34, B.14], which is a fairly new feature and a more efficient reduction technique than traditional ones using lots of shared memory, but thinking that the cuBLAS implementation does not also use that functionality would be presumptuous.

Inverse Cumulative Density Functions

The reference implementation of the scoring procedure in the clustering algorithm required some statistical functions[8] that are missing from all the frameworks, and that I have neither found any other GPGPU libraries that implements nor feel competent to implement myself. The workaround in these implementations is to delegate that specific part of the calculation to the CPU, but it requires some parameters that is computed on the GPU, so this approach

7. The actual use case is a feature-wise covariance, where the edge dimensions are below 10 and the common dimension is in the order of thousands and upwards.
8. Specifically MATLAB functions [28, `chi2inv()`] and [28, `norminv()`].

impose some additional memory transfers per cluster splitting.

In this particular case the arguments needed by the CPU are just the statical models, which results in much less than a kilobyte of data, so this additional overhead is not a big problem, and its total performance cost is worth far less than the effort of reimplementing these procedure by hand. MATLAB uses it own CPU implementation, Python implementations have one implemented in the SciPy libraries, and the C/C++ version uses an implementation from the PROB[5] library.

This idea of falling back to computing missing functionality on the CPU will always be a considerable trade-off to reimplementing it yourself, but if it imposes large data transfers it becomes very unfeasible. The fact, shown in Figure 4.6, that some GPU implementations surpass the multicore CPU MATLAB version by a long shot, indicates that need to delegate some work back to the CPU does not have to mean the end of the world.

5.4.2 C++ Templates + CUDA

C++ templates is a very expressive language feature, and it is this feature that some of the frameworks utilize to create metaprogramming-like constructs that translates into specialized kernel code at compile time. This leads to constructs that makes it possible to create very optimized GPU code without the need for low-level GPU code, but it brings with it some caveats.

First of all, anything that is going to compile CUDA kernel code requires a toolchain that is compatible with CUDA compilation. That means that template libraries that generate kernel code at compile time (e.g. Thrust and mshadow) is going to be CUDA source code, not C++, and there are some build systems (e.g. GNU Autotools) that are not directly compatible with CUDA source code. VexCL works around this problem by JIT-compiling the kernel code, so it is a native C++ framework, but instead it requires access to the CUDA compiler at runtime, but that is not such a big problem, because most installations of the CUDA runtime environment is usually accompanied with the development platform.

The commonality between these frameworks are that they create very complex recursive templates. Such templates takes annoyingly long time to compile[9] small applications, such as the streaming pipeline implementation, so I can only fear how they are to work with on large projects. The deeply nesting brings

9. Streaming pipelines implementations, ArrayFire CUDA (C++): 0.420s, CUDA: 3.984s, Thrust (CUDA template): 9.823s and VexCL (C++ template): 20.016s

with it inherent development complications because their definitions becomes unmanageable. Whenever a compiler error concerns on of these templates, its expanded name includes the entire stack of recursion, which results in names that commonly span kilobytes of data. Manually extracting information from these kinds of names is a cumbersome process, but Integrated Development Environments (IDEs) helps mitigate such problems. However, the assortment of IDEs that are compatible with CUDA C is fairly limited, and, in any case, there are other places, like debuggers and profiling utilities, where you also have to deal with these names.

6

Concluding Remarks

This thesis has presented a semester's worth of GPGPU framework exploration, experimentation and analysis. The fundamental idea was to try to uncover existing facilities for lowering the initial threshold to overcome in order to make any use of GPU hardware, which has the potential of making these computational beasts more readily available to regular developers. Rather than starting from scratch, by creating yet another underdeveloped framework suited for any one, or a few, particular usage scenarios, this thesis explores a tiny, but relevant, subset of the already existing frameworks.

Github[1] lists in excess of 5000 projects associated with CUDA and 3000 associated with OpenCL, so there is an abundance of possibilities, although many of these project are more or less identical forks of each other. Mapping this entire ecosystem would be a tremendous task, and this project only has the resources to explore a tiny subset of the ones that are available. Table 3.2 catalogs those frameworks this project has found to be the most the relevant, but, of course, this is far from an exhaustive list of neither what potentially exists out there, nor all the frameworks discovered during the course of this project.

One thing is to survey the field, but this thesis aims a bit higher than that, and Chapter 4 described a set of real SAR processing problems that have been reimplemented, and thoroughly benchmarked, in the most promising subset of

1. `https://github.com/`, a well known free hosting service for publicly available source code repository.

the surveyed framework. The idea behind these benchmarks is to get a feel for how the different frameworks aid the development process, and to determine what overhead their abstraction adds to the computational solutions.

There are two separate benchmarks, one small data intensive algorithm, and a much more extensive and compute intensive algorithm. They are supposed to represent two common types of problems, and are designed to stress different parts of the frameworks. The smaller one is implemented by frameworks that provide:

- Compile-time GPU code generation (OpenACC[37], Thrust[35], VexCL[11]).
- Extensive libraries of general-purpose GPU code (ArrayFire[45], MATLAB[28]).
- High-level language support (ArrayFire Python, MATLAB, PyCUDA[24]).

Missing functionality in some frameworks, and lack of time, limit the larger benchmark to only being implemented using ArrayFire, ArrayFire's Python bindings and MATLAB. Additionally, both benchmarks have also been implemented in low-level CUDA code, to have a sort of performance baseline to compare against the frameworks.

Experiments show that MATLAB implementations are consistently performing worse than the other implementations, and using interpreted languages in general seems to add a bit of extra overhead. In the data-intensive benchmark, every implementation performs considerably better than multi-core CPU implementations, but in the compute-intensive benchmark, only the low-level baseline implementation performs significantly better than the CPU implementation. However, optimizing the computational hot-spot in the ArrayFire implementation, by delegating that part of the algorithm to the low-level implementation, yields performance similar to that of the baseline implementation.

As for the lowering the GPGPU learning threshold, developing low-level CUDA code is complicated, because the parallel nature of it makes it hard to wrap your head around what the code is actually doing, which makes debugging a bit of a hassle. However, by itself CUDA provides a very useful C/C++ interface, and it is much easier to use than any of its predecessors (e.g. Brook). Unfortunately, CUDA is not compatible with CPU libraries, so, if you need some features from your favorite library, you might need to reimplement those features from scratch in CUDA.

Having libraries of GPU code that simply work, although not always that efficiently, is a very useful starting point, because building upon blocks that presumably work, alleviates the debugging process during any application's

development phase. If you are not comfortable with writing C/C++-code, then Table 3.2 lists GPU bindings for many languages, but you should be aware that they probably impose some overhead, and many of them require you to write the GPU code in the low-level language anyways. The code generators have tremendous potential in the form of, among others things, expressibility and portability, but with great power comes great responsibility, and development in these kinds of frameworks happens at an abstraction level that might make the code harder to debug and work with.

6.1 Conclusion

Going forward, if I were to implement some new GPU application, I would start by writing it in ArrayFire, because that would hide the parallelism, which makes the initial development process considerably easier. Then, if this implementation is performing inadequately, I would optimize parts of it, or even reimplement it bit by bit, in a low-level GPGPU interface, which, by already having a working application, is much easier than doing it from scratch.

A similar approach is doable from within MATLAB, which was the preferred language for the physicists mentioned in Chapter 1, but its low-level bindings are more cumbersome to work with, and gives you less control over memory management. That being said, ArrayFire, albeit C++, might be a considerable alternative, because it has a programming paradigm very similar to that of MATLAB, and both support more or less the same GPU functionality.

6.2 Future Work

This thesis only explored in detail a tiny subset of the available GPGPU frameworks, so there are lots of other frameworks that might provide equivalent, or even better, abstractions to work with. And, in any case, many of the explored frameworks are under active development, and similar frameworks keep popping up, so there is no telling what features the future holds.

The project had a fairly limited timeframe from the beginning, so it focused the scope onto CUDA fairly early, because that seemed to be the ecosystem that currently has the best spread of frameworks. However, the multi-platform OpenCL has a much greater potential, so over time, developers will probably shift from the CUDA ecosystem over to OpenCL, and then, or even before that, future work can explore its realm of frameworks for similar, and hopefully better, abstractions.

Source Code

Source code for the implementations along with benchmark results and a digital version of this document can be found at:
`https://static.johslarsen.net/uit/master-thesis/`

Bibliography

[1] Advanced Micro Devices, Inc: *Bolt Documentation*, 2014. https://hsa-libraries.github.io/Bolt/html/, visited on 2016-05-18.

[2] AMD: *Amd "close to metal"™ technology unleashes the power of stream computing*, November 2006. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html, visited on 2007-02-09.

[3] AMD: *Amd drives adoption of industry standards in gpgpu software development*, August 2008. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~127451,00.html, visited on 2008-12-04.

[4] Ian Buck *et al.*: *Brook for gpus: stream computing on graphics hardware*. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[5] John Burkardt: *PROB — Probability Density Functions*, 2013. http://people.sc.fsu.edu/~jburkardt/cpp_src/prob/prob.html, visited on 2016-05-31.

[6] Manuel MT Chakravarty *et al.*: *Accelerating haskell array codes with multicore gpus*. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[7] Shuai Che *et al.*: *Accelerating compute-intensive applications with gpus and fpgas*. In *Application Specific Processors, 2008. SASP 2008. Symposium on Application Specific Processors*, pages 101–107. IEEE, 2008.

[8] Tianqi Chen *et al.*: *mshadow: Matrix Shadow*, 2016. https://github.com/dmlc/mshadow/, visited on 2016-05-18.

[9] Francisco Chinchilla *et al.*: *Parallel n-body simulation using gpus*. Department of Computer Science, University of North Carolina at Chapel Hill,

http://gamma. cs. unc. edu/GPGP, Technical Report TR04-032, 2004.

[10] George Chrysos: *Intel® xeon phi™ x100 family coprocessor — the architecture*, November 2012. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, visited on 2016-05-14.

[11] Denis Demidov: *VexCL documentation*, 2016. http://vexcl.readthedocs.io/en/latest/, visited on 2016-05-17.

[12] Anthony P Doulgeris and Torbjørn Eltoft: *Scale mixture of gaussian modelling of polarimetric sar data*. EURASIP Journal on Advances in Signal Processing, 2010:2, 2010.

[13] Anthony Paul Doulgeris: *A simple and extendable segmentation method for multi-polarisation sar images*. 2013.

[14] Zhe Fan *et al.*: *Gpu cluster for high performance computing*. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[15] Giorgio Franceschetti and Riccardo Lanari: *Synthetic aperture radar processing*. CRC press, 1999.

[16] Mark Harris *et al.*: *Hemi: Simpler, More Portable CUDA C++*, 2016. https://github.com/harrism/hemi, visited on 2016-05-23.

[17] Kenneth E Hoff III *et al.*: *Fast computation of generalized voronoi diagrams using graphics hardware*. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286. ACM Press/Addison-Wesley Publishing Co., 1999.

[18] Tim Holy *et al.*: *CUDArt — Julia wrapper for CUDA runtime API*, 2016. https://github.com/JuliaGPU/CUDArt.jl, visited on 2016-05-19.

[19] Marco Hutter and Samuel Cozannet: *jcuda.org — Java bindings for CUDA*.

[20] IEEE and The Open Group: *POSIX.1-2008*, 2013. http://pubs.opengroup.org/onlinepubs/9699919799/, visited on 2016-05-05.

[21] John: *Exciting updates from accelereyes*, December 2012. http://arrayfire.com/exciting-updates-from-accelereyes/, visited on 2016-05-18.

[22] Khronos OpenCL Working Group: *The OpenCL Specification — Version: 1.0*, 2008. https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf, visited on 2016-04-28.

[23] Khronos OpenCL Working Group: *The OpenCL Specification — Version: 2.2*, 2016. https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf, visited on 2016-05-13.

[24] Andreas Klöckner *et al.*: *Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation*. Parallel Computing, 38(3):157–174, 2012.

[25] Andreas Klöckner: *Welcome to PyCUDA's documentation!*, 2016. https://documen.tician.de/pycuda/, visited on 2016-05-28.

[26] Jens Krüger and Rüdiger Westermann: *Linear algebra operators for gpu implementation of numerical algorithms*. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003.

[27] E Scott Larsen and David McAllister: *Fast matrix multiplies using graphics hardware*. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55. ACM, 2001.

[28] The MathWorks, Inc: *MATLAB — The Language of Technical Computing*, 2016. https://se.mathworks.com/help/matlab/index.html, visited on 2016-05-05.

[29] NVIDIA: *Tesla K20 GPU Active Accelerator*, November 2012. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v02.pdf, visited on 2016-04-28.

[30] NVIDIA: *CUDA Toolkit v7.5 — cuBLAS*, 2015. http://docs.nvidia.com/cuda/cublas/, visited on 2016-04-28.

[31] NVIDIA: *CUDA Toolkit v7.5 — Floating Points and IEEE 754*, 2015. http://docs.nvidia.com/cuda/floating-point, visited on 2016-05-25.

[32] NVIDIA: *CUDA Toolkit v7.5 — NVIDIA CUDA Runtime API*, 2015. http://docs.nvidia.com/cuda/cuda-runtime-api/, visited on 2016-05-17.

[33] NVIDIA: *CUDA Toolkit v7.5 — Profiler User's Guide*, 2015. http://docs.nvidia.com/cuda/profiler-users-guide/, visited on 2016-05-28.

[34] NVIDIA: *CUDA Toolkit v7.5 — Programming Guide*, 2015. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, visited on 2016-04-28.

[35] NVIDIA: *CUDA Toolkit v7.5 — Thrust*, 2015. http://docs.nvidia.com/cuda/thrust/, visited on 2016-05-17.

[36] NVIDIA Research: *CUB Documentation*, 2016. https://nvlabs.github.io/cub, visited on 2016-05-17.

[37] OpenACC-Standard.org: *The OpenACC Application Programming Interface — Version 2.5*, October 2015. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, visited on 2016-05-19.

[38] OpenMP Architecture Review Board: *OpenMP Application Programming Interface — Version 4.5*, November 2015. http://www.openmp.org/mp-documents/openmp-4.5.pdf, visited on 2016-05-19.

[39] David Andrew Patterson and John LeRoy Hennessy: *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 4th edition, 2009, ISBN 0-12-374493-7.

[40] PCI-SIG: *PCI Express Base Specification, Revision 2.1*, March 2009.

[41] Python Software Foundation: *Python 3.5.1 Documentation*, 2016. https://docs.python.org/3/index.html, visited on 2016-05-28.

[42] Christoph Rohland, Hugh Dickins, and KOSAKI Motohiro: *Tmpfs is a file system which keeps all files in virtual memory.*, 2010. https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt, visited on 2016-05-05.

[43] Michel Steuwer: *SkelCL — A Skeleton Library for Heterogeneous Systems*, 2015. http://skelcl.uni-muenster.de/, visited on 2016-05-17.

[44] Theano Development Team: *Theano: A Python framework for fast computation of mathematical expressions*. arXiv e-prints, abs/1605.02688, May 2016. http://arxiv.org/abs/1605.02688.

[45] Pavan Yalamanchili *et al.*: *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. Atlanta, 2015. https://github.com/arrayfire/arrayfire.