A Framework for Composing High-Performance OpenCL from Python Descriptions

Michael Anderson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-210
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-210.html

December 5, 2014

A Framework for Composing High-Performance OpenCL from Python Descriptions

by

Michael Jeffrey Anderson

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt Keutzer, Chair
Professor James Demmel
Associate Professor Sara McMains

Fall 2014

A Framework for Composing High-Performance OpenCL from Python Descriptions

Abstract

A Framework for Composing High-Performance OpenCL from Python Descriptions

by

Michael Jeffrey Anderson

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt Keutzer, Chair

Parallel processors have become ubiquitous; most programmers today have access to parallel hardware such as multi-core processors and graphics processors. This has created an implementation gap, where efficiency programmers with knowledge of hardware details can attain high performance by exploiting parallel hardware, while productivity programmers with application-level knowledge may not understand low-level performance trade-offs. Ideally, we would like to be able to write programs in productivity languages such as Python or MATLAB, and achieve performance comparable to the best hand-tuned code.

One approach toward achieving this ideal is to write libraries that get high efficiency on certain operations, and call these libraries from the productivity environment. We propose a framework that addresses two problems with this approach: that it fails to fuse operations for efficiency, and that it may not consider runtime information such as shapes and sizes of data structures. With our framework, efficiency programmers write and/or generate customized OpenCL snippets at runtime and the framework automatically fuses, compiles, and executes these operations based on a Python description.

We evaluate the framework with case studies of two very different applications: space-time adaptive radar processing and optical flow. For a space-time adaptive radar processing application, our framework's implementation is competitive with a hand-coded implementation that uses a vendor-optimized library. For optical flow, a computer vision application, the framework achieves frame rates that are between 0.5× and 0.97× hand-coded OpenCL performance.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express thanks to Kurt Keutzer for advising me and providing encouragement and direction. I would also like to thank the rest of my committee: Jim Demmel, Kathy Yelick, David Wessel, and Sara McMains.

I received a great deal of help and support from current and former members of the PALLAS group, including Jike Chong, Bryan Catanzaro, Mark Murphy, Bor-Yiing Su, Narayanan Sundaram, David Sheffield, Katya Gonina, Patrick Li, Peter Jin, Forrest Iandola, Zach Rowinski, and Matt Moskewicz. Specific thanks to David Sheffield for his close collaboration on several projects and daily interactions in the lab. Also to Bryan Catanzaro for helpful discussions and for sharing his enthusiasm for GPU computing. Bryan's work on the Copperhead framework laid the foundation for the framework described in this dissertation. Katya Gonina's work on PyCASP connected the ideas of SEJITS, structural patterns, and composition, which helped to motivate and organize my framework. Thanks to Narayanan Sundaram for frequent discussions. Narayanan's work on dense optical flow led to my focusing on this application as a case study in this dissertation. Finally, to Patrick Li for helping me with my dissertation talk.

I also want to acknowledge members of the BeBOP group. Thanks to Vasily Volkov for helping me understand the GPU register file and encouraging me to work on small batch factorizations. Also to Grey Ballard for helping me understand communication-avoiding algorithms and how to map them to cache hierarchies. Finally, to Erin Carson and Nick Knight for their help with communication-avoiding linear solvers.

Chapter 1

Introduction

The transistor density of integrated circuits doubles every 18 to 24 months in a trend that is commonly referred to as Moore's law. Since its initial observation in 1965 [1], this exponential growth has continued steadily, enabling new applications with cheaper and more powerful processors. In the 28 years from 1982 to 2010 the number of transistors on Intel's processors grew from 134,000 transistors for the 80286 to 1.17 billion transistors for the Core i7. [2]. This is an 8731× increase that corresponds roughly to 13 transistor-count doublings.

Single-thread performance, as measured by the SPEC benchmarks [3], has also historically increased at an exponential rate similar to the Moore's law trend, growing 25% per year from 1978 to 1986, 52% per year from 1986 to 2003, and 22% per year from 2003 to 2010 [2]. This exponential speedup has been attributed to faster transistors, larger on-chip caches, and clever micro-architectural techniques enabled by increased transistor budgets [4]. In the mid-2000s however, the growth of clock frequencies of commercial processors flattened, and the increases in single-thread performance slowed [5, 6]. The scaling technique that allowed transistors to shrink while maintaining the same overall power, called Dennard scaling [7], came to an end and power consumption became the limiting constraint in microprocessor design [4, 8]. Since then, the extra transistors provided by the continuation of Moore's law have been largely devoted to multi-core parallelism, wider SIMD units, larger caches, and further integration.

As hardware becomes more parallel, it becomes harder to get high performance on today's architectures. Achieving high performance can require significant effort and understanding of low-level hardware details. Most programmers are focused on their application and are not interested in or perhaps not capable of tuning low-level code to target architectures. They would rather program in productivity languages such as MATLAB, Python, or Perl, and call optimized libraries in a sequential manner. These are called *productivity programmers* [9]. Those who implement the optimized libraries are called *efficiency programmers*. Efficiency programmers write using low-level languages, and use tools such as OpenMP, vector intrinsics, and OpenCL. They are experts in achieving high hardware utilization, throughput, and maximizing reuse in caches.

The *implementation gap* [9, 10] is a term for the gulf between the productivity program-

mers and the efficiency programmers. This gap presents a serious problem for developers of emerging applications, and also for chip makers who see both performance and programmability as key selling points of their products. In a typical scenario today, a productivity programmer will write an application in MATLAB or Python, then send the specification to the efficiency programmer to implement as an efficient library. The productivity programmer may make high level decisions without knowledge of the low-level parallel performance implications. Similarly, the efficiency programmer may expend effort optimizing computations that can easily be eliminated with a slight change in the high-level approach. Bridging the implementation gap is necessary to avoid these undesirable outcomes.

On one end of the implementation gap is the efficiency programmer. Chapters 3 and 4 describe the efficiency-level implementation of dense linear algebra applications on GPUs. In Chapter 3 we optimize the QR factorization and apply it to a video background subtraction application. The matrix is very tall and skinny, meaning there are many more rows than columns. We show that a communication-avoiding QR factorization can outperform the traditional blocked QR factorization. High performance was achieved through extensive manual and auto-tuning the computation to fit the GPU hardware. In Chapter 4, we focus on solving many small (e.g. 56×56) matrix factorizations in parallel on GPUs. We build a model of the GPU memory hierarchy to inform our tuning. In the case of small matrix factorizations, we find that runtime code generation, which allows us to use the GPU register file, is essential for good performance.

We set out to build a framework to attempt to bridge the implementation gap. We use the experiences tuning dense linear algebra in Chapters 3 and 4 to determine our framework's design and requirements:

- First, our experience convinces us that transparency is necessary for high performance. That is, a framework must allow for low-level code tuning. This empowers the efficiency programmer to make problem-specific or machine-specific optimizations that our framework never may have envisioned. This is in contrast to frameworks that would choose to hide these details behind programming abstractions.

- Second, optimizing with knowledge of specific shapes and sizes of data structures was very effective in our experience. In Chapter 3, the algorithm and block sizes we chose worked particularly well for tall and skinny matrices, but less well for square matrices. For the small linear algebra operations in Chapter 4, we generate different code for different sized matrices.

- Third, we observed that the time it takes to perform computations is much less than the time it takes to load operands from memory. Even for relatively compute-intensive small dense linear algebra factorizations, described in Chapter 4, we show that roughly 25% of the time is spent loading and storing data from global memory, and only 75% of the time is spent on computation. The gap between processor performance and memory performance is a well known problem with today's machines that seems to get worse over time [11]. This leads to a situation where composing two operations

can provide substantial speedups if the operations access the same data and memory traffic can be avoided. Whether composition is profitable or not can depend on the details of the operations being composed, and on machine details, as we demonstrate in the case studies in Chapters 6 and 7.

- Finally, we recognize that in order to reach productivity programmers, we must provide a front end in a popular productivity language such as Python or Ruby.

In light of these observations, we propose a framework called *Hindemith*, described in Chapter 5, that uses the selective embedded just-in-time specialization (SEJITS) methodology [9], along with high-performance hand-coded operations, organized around the principles of pattern-based software engineering. Inspired by recent SEJITS frameworks, Copperhead [12], PyCASP [13], and Three Fingered Jack [14], our framework is embedded in Python, meaning that productivity programmers can invoke it by writing specialized Python functions. The productivity programmer designs a program using a hierarchical composition of structural patterns and specialized operations. Any operation that is invoked by the productivity programmer must have a corresponding specialized implementation, which is written by the efficiency programmer. The entire program, including each individual operation, is processed at runtime, fused together, and just-in-time compiled using OpenCL. Furthermore, the shapes and sizes of all the data structures are propagated through the program description. This means that each operation has access to this vital runtime information during the code generation phase.

Hindemith is designed to be transparent, meaning that the efficiency programmer can provide operations with as much control over the hardware as OpenCL provides. The space-time adaptive processing case study in Chapter 6 shows the benefit of this transparency as it allows operations to be specifically tuned to run in the GPU's register file. The framework also provides automatic fusion of related operations. The optical flow case study in Chapter 7 is an example where extremely light-weight operations, such as vector addition, are combined into one to reduce unnecessary traffic between processors and memory.

We perform two case studies in Chapters 6 and 7, implementing a space-time adaptive processing (STAP) application and optical flow application in our framework. The STAP application boils down to doing many small linear algebra operations in parallel, so we leverage the hand-coded solutions developed in Chapter 4 and wrap these as framework operations. On an Intel Core i7 CPU, our STAP implementation exceeds the performance of a serial C reference implementation by over 10×. On an NVIDIA GT 640 GPU, our implementation is 8% slower than a CUDA + CUBLAS version, but 2.4× faster than the CUDA + CUBLAS version if only considering the time spent on computation. We also find that excessive kernel fusion can hurt performance for the STAP application. In Chapter 7, we implemented two different optical flow methods and four different linear solvers used for optical flow in our framework. Compared to ArrayFire, a commercial JIT-compiled linear algebra library for GPUs, our solutions performed between 1.4× to 2× faster on the NVIDIA GT 640 platform, and between 1.8× and 3.5× faster on the Intel Core i7 platform.

On the same benchmarks, the framework achieved between 0.5× and 0.97× the performance of our hand-coded implementation. These performance comparisons do not include JIT-compilation time. The applications we chose are meant to be run repeatedly in a real-time setting, and JIT-compilation time only appears in the first run of the program.

1.1 Thesis Contributions

This thesis makes the following contributions:

- In an effort to bridge the implementation gap, we propose a framework called Hindemith (Chapter 5) that is transparent and allows for custom hand-tuned code, can take advantage of runtime knowledge such as shapes and sizes, composes operations to reduce communication between processors and memory, and is embedded in a productivity language. To this end, we deliver as a software artifact an implementation of Hindemith. The Hindemith framework uses the selective embedded just-in-time specialization (SEJITS) methodology, along with high-performance hand-coded operations, organized around the principles of pattern-based software engineering.
- We evaluate the Hindemith framework using two application case studies: space-time adaptive processing (STAP) and optical flow. Ignoring JIT compilation time, our framework delivers optical flow performance that is between 0.5× and 0.97× our own hand-coded implementations. For STAP, the our implementation is competitive with a hand-coded CUDA + CUBLAS implementation.

1.2 Thesis Outline

- Chapter 2 covers background on the parallel programming problem, parallel platforms, and software.
- Chapter 3 demonstrates efficiency-level programming and tuning for the communication-avoiding QR factorization targeting GPUs.
- Chapter 4 demonstrates efficiency-level programming and tuning for solving small linear algebra factorization on GPUs. We also introduce a predictive model of the GPU memory hierarchy.
- Chapter 5 explains how the Hindemith framework works, including how to create operations, the analysis performed by the framework, and how it is embedded in Python.
- Chapter 6 provides a case study using Hindemith to implement a space-time adaptive processing (STAP) benchmark.

- Chapter 7 provides a case study using Hindemith to implement the optical flow application.
- Chapter 8 concludes.

Chapter 2

Background and Motivation

In this chapter, we provide relevant background material. We begin in Section 2.1 by introduce the parallel hardware targets used in the rest of this thesis. In Section 2.3 we describe the parallel programming problem that arises as a result of parallel hardware, and how we must balance productivity, efficiency, and portability. Current solutions to the parallel programming problem are described in Section 2.4, including efficiency languages, productivity languages and domain-specific languages (DSLs). In Section 2.6, we give an overview of parallel programming patterns and SEJITS. These are concepts that serve as the basis for our framework, which is introduced in Chapter 5.

2.1 Parallel Hardware

Parallel processors have become ubiquitous. A personal computer today is likely to include a single chip containing a quad-core CPU with 256-bit AVX 2.0 SIMD extensions, and an integrated GPU [15]. One or more discrete GPUs can be added to the system as well to create a commodity high-performance heterogeneous parallel machine. We focus our attention on two design points for parallel systems: graphics processors (GPUs) and general-purpose multicore processors (CPUs). We chose these particular design points simply due to their popularity; most desktops and laptops sold today include programmable multicore and GPU hardware.

Graphics Processors (GPUs)

Graphics processors (GPUs) were first designed as mostly fixed-function accelerators for graphics applications, but have since been further developed into more general-purpose programmable data-parallel processors capable of a wide variety of highly parallel workloads [16, 17]. In order to get high performance on a GPU, an application must provide a large amount of parallelism to fill the GPU with work and to keep the memory system busy. Applications that are dense or structured are better suited to GPUs than sparse irregular

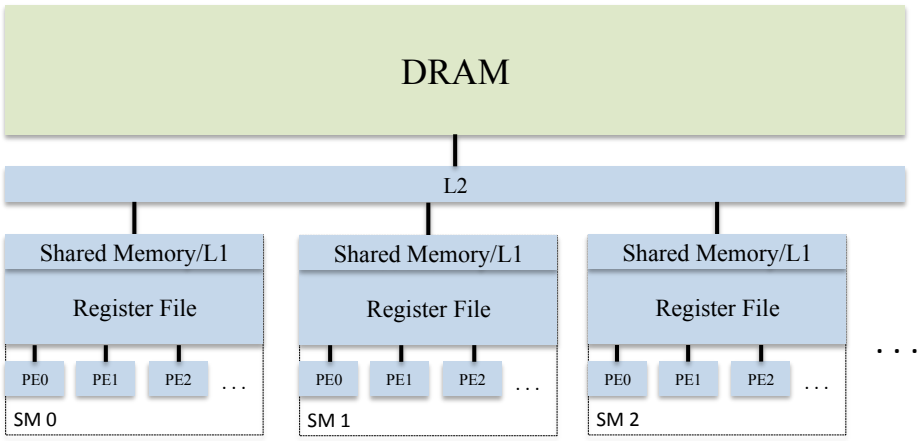


Figure 2.1: Block diagram of the memory hierarchy for an NVIDIA Fermi GPU

applications, because the GPU hardware performs better when it can coalesce parallel work into SIMD instructions and wide memory accesses.

A block diagram of the memory hierarchy for an NVIDIA Fermi GPU is shown in Figure 2.1. The global dynamic random access memory (DRAM) is located on the PCI card in the case of a discrete GPU. The Fermi architecture uses multiple GDDR5 DRAM interfaces optimized for fast parallel access [18]. The 768 KByte L2 cache is shared among all streaming multiprocessors (SMs). Each SM has a 64 KByte scratchpad memory that can alternatively be configured as an L1 cache. The register file, as suggested by the figure, is the largest on-chip memory with 128 KByte per SM. An important feature of the GPU compared to multicore CPUs is the structure of the memory hierarchy. On the Fermi GPU, the largest cache memories are physically closest to the processing elements. On CPUs the cache memories are much larger per core then on GPUs, and the larger CPU caches are physically further away from the processing elements than the smaller CPU caches.

GPUs are also being integrated onto the same die as CPUs. Recent offerings from Intel (e.g. Haswell with HD integrated graphics) and AMD (e.g. Fusion Llano) have included increasingly powerful and programmable GPUs [19, 20, 21]. The integrated GPUs along with the host CPUs are also programmable with OpenCL using the Intel OpenCL SDK and the AMD APP SDKs respectively. The advantage of providing integrated GPUs is that one less ASIC is required for a complete system, as compared to discrete GPUs.

Multicore Processors

Unlike GPUs, multicore processors are designed to perform well on a wide variety of general purpose workloads including desktop processing, games, web servers, and scientific computing. Among other techniques, multicore processors rely on deep cache hierarchies to increase performance by making main memory appear closer and faster than it is. Cache memories

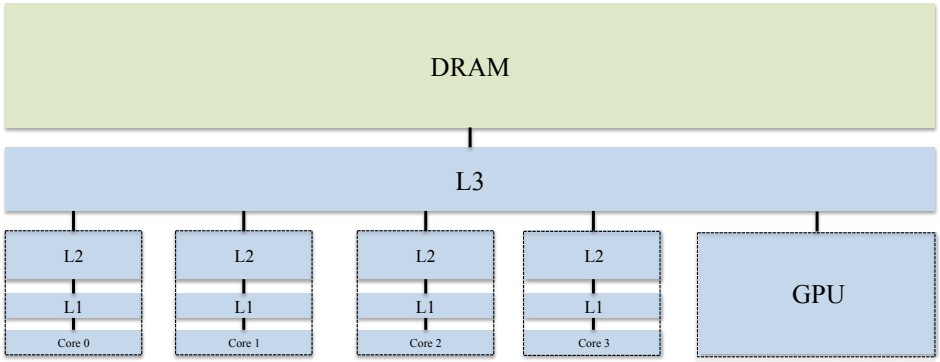


Figure 2.2: System block diagram for an Intel Haswell quad-core CPU with integrated graphics

were already common in "virtually all modern large computer systems" by 1982 [22], and recent Intel processors include up to four levels of cache [19]. Multicore processors also typically have dedicated hardware for single-instruction, multiple-data (SIMD) instructions that take advantage of parallelism and amortization of instruction fetch overhead to increase performance on data-parallel workloads [23, 24]. In recent years, SIMD units have increased in width and added support for new operations such as double-precision floating-point, fused multiply-add, and scatter/gather instructions [25].

Figure 2.2 shows a system block diagram for the CPU portion of an Intel Haswell quad-core processor [19, 20]. The system includes four cores, which all share an 8MB L3 cache memory. Each core has a 256 Kbyte L2 and a 64 KByte L1 cache, split evenly between instructions and data, per core. The system maintains coherence between cache memories. Some processor configurations include an integrated GPU, which also has access to the L3 cache memory.

The growth in SIMD width and throughput, as well as core count, has increased the performance of scientific and throughput-oriented workloads on multicore processors. A recent study has shown that, by making use of SIMD and other architectural features, the performance gap for high-throughput workloads between multicore processors and GPUs is 2.5× on average [26]. The memory bandwidth gap between multicore processors and GPUs may be diminishing as well, with new L4 cache memories achieving peak bandwidth speeds of 102 GB/s [19, 20].

2.2 The Memory Wall

Another major trend impacting performance is the growing disparity between processor speed and memory speed, often called the *memory wall* [27]. The observation in 1994 that DRAM speeds were increasing by only 7% per year and processor performance was increasing at a rate of 80% per year indicated that average memory access time would, at some point,

dominate the runtimes of many important applications. The STREAM benchmark, released in 1995, measures performance on a set of bandwidth-bound vector operations such as vector addition [28]. The STREAM benchmark is useful not only for characterizing a single machine, but also observing trends in memory bandwidth performance over time. Wulf and McKee's presentation of STREAM benchmark data shows that from 1980 to 2000, memory bandwidth increased at roughly a 35% annual rate, while peak processor floating-point performance increased at roughly a 50% annual rate [11]. Memory latency improves at even a slower rate than bandwidth. An analysis of memory performance from 1982 to 2001 showed that memory bandwidth had grown from 13 MByte/s to 1,600 MByte/s while memory latency had only improved from 225 nanoseconds to 52 nanoseconds [29]. They observe that for every doubling in memory bandwidth, latency has only improved by roughly a factor of 1.2 to 1.4. These trends are expected to continue in the near term. A recent report on trends in supercomputing estimated the annual improvement in single-chip floating-point performance to be 59% through year 2020, compared to an annual 25% improvement in DRAM bandwidth and an annual 5.5% improvement in DRAM latency [30]. The high energy cost of moving data compared to performing computation is also a concern. According to another recent report, it costs roughly half the energy to fetch three double-precision operands as it does to perform a multiply-add operation on those operands from on-chip memories that are 1 millimeter away, and it costs an order of magnitude more energy to fetch from 20 millimeters away than to perform the multiply-add [31].

These memory trends affect some computations more than others. Certain applications that can make effective use of cache hierarchies are less affected by poor memory performance than applications that must stream large amounts of data to and from global memory. This has motivated research into *communication-avoiding algorithms* that aim to turn communication-bound applications into compute-bound applications by asymptotically reducing the amount of data transferred between levels of the memory hierarchy, and also asymptotically reducing the number of synchronizations across parallel elements [32, 33]. Communication-avoiding algorithms have the potential to provide speedups on existing applications, and may also improve energy efficiency in future microprocessor systems [4].

2.3 The Parallel Programming Problem

The ubiquity of parallel hardware brings a problem: how can we enable programmers to take advantage of it? Parallel hardware can enable higher efficiency through the use of multiple cores or multiple SIMD lanes. However, to make use of parallel resources, the programmer must identify parallelism in his or her application and modify the implementation for parallel execution. This can reduce programmer productivity by adding complexity to the specification. Additionally, the programmer may want to tune his or her implementation for particular hardware details for maximum efficiency, which reduces productivity and potentially costs portability as well. The programmer thus faces a trade-off between competing forces of efficiency, portability, and productivity illustrated by the three axes in

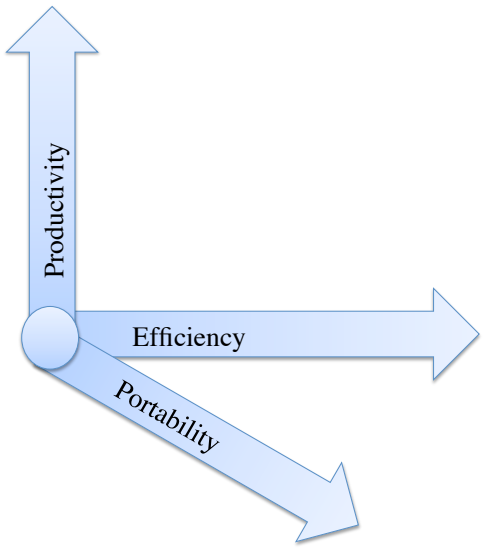


Figure 2.3: The parallel programming problem entails a trade-off between efficiency, portability, and productivity.

Figure 2.3. The next subsections will describe the competing forces of efficiency, portability, and productivity in more detail.

Efficiency

We measure efficiency in different ways depending on the application and its constraints. Floating point applications and benchmarks, such as LINPACK [34], have historically been measured in terms of floating-point operations completed per second (FLOPS). This work focuses on machines capable of computation on the order of billions of floating-point operations per second, or GigaFLOPS (GFLOPS). Bandwidth performance is reported in terms of billions of bytes per second transferred from memory to processors (GBytes/s). Power is measured in Watts (W), with a typical machine drawing between 10W and 300W in total wall power. Energy consumption, measured in Joules (J), is an important application performance measure. We also report energy efficiency of floating-point applications in terms of GigaFLOPS per Watt of wall power (GFLOPS/W), which can be alternatively interpreted as the total number of floating-point operations per Joule of energy spent. A typical GFLOPS/W number in today's technology would be 1.0, possibly for a machine that draws 100 Watts and can perform a matrix-multiplication operation at a rate of 100 GFLOPS.

A given application's performance is often contextualized by reporting GFLOPS or GBytes/s achieved as a percentage of the peak attainable GFLOPS or GBytes/s. The roofline model is a methodology for identifying whether an application is bounded by floating-point throughput or memory bandwidth, which can suggest the most profitable optimization techniques to apply in each case [35]. Getting a high percentage of peak performance often

requires significant hand-tuning and knowledge of underlying hardware. A study of sparse matrix-vector multiplication for parallel processors identified optimizations such as loop unrolling, software pipelining, and register blocking, that helped to increase the application's bandwidth utilization [36]. In Chapters 3, 4, and 6 of this work, we describe in detail the hand tuning required to increase performance of certain dense linear algebra operations on GPUs.

Portability

We would like to have the same code run on multiple processor designs. Often, a specific architecture will require a specific programming environment in order to program it. Multicore SIMD units, for example, often require special instructions, which can be accessed using assembly intrinsics. On GPUs, the instruction set is kept hidden from the programmer to allow for greater hardware flexibility. The programmer must use special languages such as CUDA or OpenCL to program GPUs. These languages are designed to encourage the use of fine-grained parallelism, which is necessary to achieve high performance on GPUs.

Industry recently coalesced around a new open standard for heterogeneous parallel computing called OpenCL (Open Computing Language) [37, 38]. Programs can be written once in the OpenCL language and run on a wide range of parallel devices, providing they support the standard. Programs are typically compiled at runtime to enable such portability. OpenCL is now supported on a wide range of devices including GPUs, multicore CPUs and their integrated graphics, accelerators, DSPs, and FPGAs. However, in many cases OpenCL programs are not performance portable, meaning specific tuning needs to take place for each potential OpenCL target in order to achieve good performance.

Productivity

Productivity is a measure of how challenging and/or time consuming it is to implement an application. This can roughly translate into number of lines of code or number of hours spent on implementation and maintenance. Languages such as Python or Perl are thought to be more productive than languages such as C++ because they are interpreted and untyped. Third party libraries can also increase the productivity of a programming environment. Python, in particular, has an extensive catalog of third-party libraries ranging from numerical processing [39, 40] to data visualization [41], to posting messages on Twitter [42].

Programming environments can be more productive especially when they are designed for a specific domain. For example, MATLAB [43] is a domain-specific programming environment for scientists and engineers that allows the programmer to very succinctly describe matrix and vector codes, image and signal processing applications, statistics applications, etc. Tools such as AMPL [44] or CPLEX [45] are very productive for modeling and solving a particular class of problems including linear programming, mixed integer programming and convex optimization.

2.4 Current Solutions

Programmers have many choices when it comes to programming parallel hardware. We outline them here in three broad classes: productivity languages, efficiency languages, and domain-specific languages (DSLs).

Productivity Languages

Productivity languages were designed with programmer productivity in mind. Languages such as MATLAB, Python, and Perl are among the most popular productivity languages. They are often not compiled and use interpreters in order to execute. This leads to poor performance which, in the case of Python, can be improved through the use of JIT compilers (PyPy [46]) or C language extensions (Cython [47]). Productivity languages are often untyped, which contributes to their slow performance. They often come with extensive debugging support, and also a wide array of supporting libraries. MATLAB has over 40 toolboxes available for purchase in domains ranging from symbolic math to bioinformatics [48], and untold numbers of 3rd party functions written in efficiency languages and wrapped using the MATLAB mex interface. Any of these 3rd party functions are free to take advantage of available parallel hardware. Python also has an extensive catalog of third-party libraries including popular tools for numerical processing [39, 40] and data visualization [41].

Efficiency Languages

As opposed to productivity languages, which are designed with programmer productivity in mind, efficiency languages are often designed with details of the target hardware in mind. The C language, which grew out of Bell Labs in the 1970s, gives the programmer a high degree of control over the machine and was useful especially for implementing the UNIX operating system [49]. The Fortran language works especially well with optimizing compilers targeting vector and parallel hardware, particularly for scientific applications with dense loop nests [50]. With the advent of highly programmable GPUs, new efficiency languages like CUDA [51] and OpenCL [38] have emerged to provide programmers with a way to write implicitly parallel code that can execute effectively on GPU hardware.

Most high-performance software is written in an efficiency language and packaged as a library, to be called either from another efficiency language or encapsulated in a wrapper and called from a productivity language. Among the most popular efficiency language libraries are OpenCV [52], a collection of computer vision routines, and Basic Linear Algebra Subprograms (BLAS) and LAPACK [53, 54], efficient and numerically correct implementations of common linear algebra operations. Hardware vendors also release versions of these libraries optimized for their hardware, for example AMD's Core Math Library (ACML), Intel's Math Kernel Library (MKL), and NVIDIA's CUBLAS.

Domain Specific Languages (DSLs)

Domain specific languages can potentially provide both high productivity and high efficiency simultaneously. This is achieved by limiting the scope of what can be implemented to a specific domain such as matrix algebra or image processing, or to a specific style of computation such as data parallel computations.

ArrayFire [55] is a commercial parallel programming framework that does just-in-time compilation of array codes to compose operations in such a way as to eliminate unnecessary reads and writes, generating OpenCL at runtime. Halide [56, 57], designed for image processing pipelines, allows the programmer to write transformations on images, along with a specification of the schedule in which the operations execute. Theano [58] takes mathematical operations specified at a high level in Python, performs both numerical and compiler optimizations, and executes on GPUs and multicore. Liszt [59] is a performance-oriented domain specific language for partial differential equation (PDE) solvers.

Instead of targeting a certain domain, some frameworks target applications that have a specific computational structure. Sequoia [60] is a framework that optimizes recursive applications and automatically tunes them to fit various memory hierarchies. Copperhead [12, 61] is a framework that takes a subset of Python, namely data parallel operations, does runtime compilation and executes these operations efficiently on GPUs and multicore. Three Fingered Jack [14, 62] is a runtime optimizing compiler targeting dense loop nests that targets CPUs, GPUs, and custom hardware. PyCASP [13, 63] is a SEJITS framework designed for content-analysis applications that leverages structural pattern insights to perform composition of specialized functions from a productivity-language specification.

There is also an effort to share and reuse DSL-building infrastructure. Delite [64] is a DSL building framework based in Scala, and OptiML [65] is an example DSL for machine learning that uses Delite. The ASP [66] framework is a tool for embedding specializers in Python using the selective embedded just-in-time specialization (SEJITS) methodology.

2.5 Limitations of Current Solutions

Each class of parallel programming solution has advantages and disadvantages. Efficiency languages provide the programmer with control over machine resources to allow for bare-metal tuning, however this style of programming is less productive than productivity languages. Productivity languages are exceedingly slow because they must interpret every operation, but one can embed libraries written in efficiency languages. This provides access to high-performance libraries from productivity languages. However, most productivity languages do not take advantage of composition opportunities. Domain specific languages can do an excellent job of composing related operations for efficiency, but these languages are often restricted to a specific sub-domain.

We begin to motivate our solution in Chapters 3 and 4 with our experiences hand-tuning dense linear algebra operations for GPUs. From these experiences, we derived the following

set of requirements for our framework:

- It is transparent and allows for custom hand-tuned code,
- It can take advantage of runtime knowledge such as shapes and sizes,
- It composes operations for efficiency, taking advantage of data reuse to reduce communication between processors and memory and
- It is embeddable in a popular productivity language.

Here we describe these requirements in more detail and the motivation behind each.

Requirement 1: Transparency

The programmer must be able to write arbitrary low-level code that takes advantage of machine details in order to attain high performance. Machine details should be transparent and not always hidden behind programming abstractions. Instead, the programmer should be able to write directly in efficiency languages if he or she chooses.

In Chapters 3 and 4, we note that programming for the memory hierarchy and register file are essential to achieving high performance. This includes blocking computations, choosing particular block sizes that fit the memory hierarchy, and application-specific optimizations such as transposing data to improve performance. This strongly influences our opinion that any solution to the parallel programming problem should be transparent and provide the efficiency programmer with the option to write custom, hand-tuned code.

Requirement 2: Knowledge of Runtime Shapes and Sizes

Code should be generated at runtime and the programmer should be able to use runtime information in the code generation process, such as datatypes, problem shapes, and problem sizes.

It is difficult to compile directly to machine code without datatype information, which is often not available until runtime when working with interpreted languages such as Python. Runtime code generation can solve this problem elegantly. Shape information can also be useful for performance optimization, and different parallelism strategies may be advantageous for differently shaped problems. In Chapter 3, we show that a different parallel algorithm, communication-avoiding QR, can outperform a more traditional algorithm, blocked Householder QR, but only when the matrix has a certain shape. In Chapter 4 and the case study in Chapter 6, we show the benefits of compiling different code for different sized problems. Runtime code generation allows us to use the GPU register file, but in order to use it we have to know the exact size of the problem. In Section 6.3, we show that this code generation and use of the register file leads to a 5.7× speedup for the system-solve kernel in the space-time adaptive processing application.

Requirement 3: Composition

A framework should have the ability to join multiple operations at runtime into a single operation, if this provides higher performance than executing the operations individually. Composition of operations is often essential for high performance. Once data has been fetched into on-chip cache memories or registers, it should stay there as long as it is being used. This is especially true for light-weight operations such as BLAS1 vector operations or 2D stencils. However, even in the case of heavy-weight operations, the time to load and store data is non-negligible. In Section 4.5, we show that when doing a set of 56×56 LU factorizations on a Quadro 6000 GPU, 25% of the time is spent loading and storing data to and from global memory, and only 75% of the time is spent on the $O(n^3)$ computation.

Given the long-term trend towards increasing processor speeds over memory speeds, described in Section 2.2, composition is becoming more and more important. In order to achieve high performance, a framework should use its runtime code generation abilities to fuse related operations into a single loop nest or OpenCL kernel. In the case of discrete GPUs with disjoint memory systems, it is also important to leave data in device memory for as long as its being used and to minimize the amount of transfers to and from the device. A framework should also manage on which device the data structures reside and try to minimize data transfers between devices, although recent CUDA releases have taken steps to unify the host and device memory spaces and manage data transfer automatically at a lower software layer [67].

Requirement 4: Embedded in a Popular Productivity Language

Python is a popular language among domain experts and productivity programmers. Making a framework programmable from Python gives productivity programmers a reasonable interface to use the framework. It also means that the inputs and outputs of the framework can easily be passed to other libraries, visualization tools, and frameworks that provide Python bindings.

2.6 Our Solution

Our solution to the parallel programming problem follows the example of prior work in pattern-oriented frameworks [10, 13, 12, 68]. We begin in this section by describing design patterns for parallel programming. These are common solutions to recurring problems that arise while engineering parallel software. They describe the software architecture, how components are arranged and how data flows between components, the computation itself, and various tools and techniques for efficient parallel implementation. The aim of the patterns is to document and convey techniques that are necessary and familiar to an expert software architect and parallel programmer.

Patterns are not, however, specific software artifacts. They are recorded in documents and books that programmers reference. Frameworks are software artifacts that can restrict

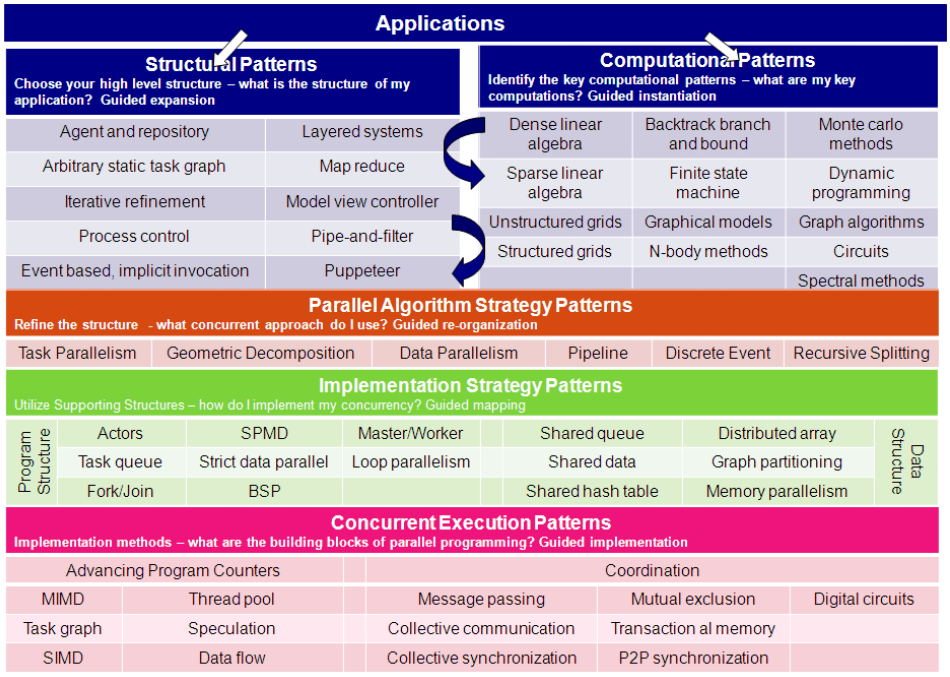


Figure 2.4: Keutzer and Mattson's Our Pattern Language [71]

the programmer to a specific set of patterns or a specific method of implementation. Selective embedded just-in-time specialization (SEJITS) is a methodology for designing frameworks that allows for runtime code generation of selected patterns and computations. We introduce the SEJITS methodology here and we describe our SEJITS framework, called Hindemith, in Chapter 5. Details of how our proposed framework differs from other related frameworks are in Section 5.2.

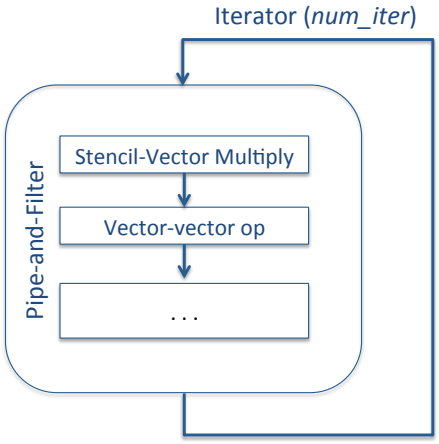
Patterns

Design patterns are solutions to recurring problems. These problem-solution pairs are documented and named as *patterns* in a *pattern language* [69]. Though design patterns were originally conceived to describe civil architecture, the concept has since been successfully applied to computer software, the most famous example being the object-oriented programming textbook *Design Patterns: Elements of Reusable Object-Oriented Software* by the so-called Gang of Four [70].

The domain of high-performance parallel software is different in many ways than traditional object-oriented software design. A pattern language specifically for parallel programming was designed to address the specific concerns of parallel programmers. A revision of this pattern language is shown in the lower half of Figure 2.4. The idea is that parallel programmers would read up on each of these patterns, in the text *Patterns for Parallel Programming* [72], which provides a step-by-step guide toward efficient parallel implementa-

```
def jacobi(D, R, b, x):
  for i in range(num_iter):
    r = b - R*x
    x = D * r
  return x
```

(a) Jacobi linear solver example written in Python



(b) Architected using structural patterns

Figure 2.5: Architecting the Jacobi linear solver with structural patterns

tions. Later, ideas from software architecture were incorporated into the pattern language. These *structural* and *computational* patterns, included in the top half of Figure 2.4, provide a common language for software architecture and computation. The final product, called *Our Pattern Language* [71] serves as the basis for our design philosophy. The following sections will describe each level of Our Pattern Language in more detail.

Structural Patterns

The process of architecting software using the pattern language begins with an iterative process of choosing a combination of structural and computational patterns are used to map out a software architecture. Structural patterns describe common ways in which software components are composed. These are like the boxes and arrows one would write on a white board to describe a software architecture. Structural patterns are useful not only to communicate a design, but because they encode modularity and certain invariants that can be exploited by frameworks. The PyCASP framework leverages three structural patterns (Pipe and Filter, Iterator, and MapReduce) for optimizations such as operation composition, allowing efficient execution of audio processing applications [13, 63].

Figure 2.5 shows Python code for a Jacobi solver on the left and its software architecture on the right. The architecture uses two structural patterns. The highest level pattern is an iterator that loops *num_iter* times. Inside the loop there is a pipe-and-filter pattern, containing a number of computations including stencil-vector multiplication and vector-vector operations.

Computational Patterns

Computational patterns describe core classes of computations found in applications [5]. Where as structural patterns describe the layout of computation, computational patterns describe what is computed and how. Computational patterns fit into a software architecture as the nodes of a graph, where the graph is a hierarchial composition of structural patterns. Computational patterns are usually supported by application-specific or pattern-specific libraries. Examples include the CombBLAS library for a certain class of graph algorithms [73], BLAS and LAPACK for the dense linear algebra pattern [53, 54], and the Liszt framework for structured grid computations [59].

Lower Levels of Pattern Language

Notice that both structural and computational patterns do not directly deal with concurrency or parallelism. After a software architecture is designed using computational and structural patterns, the lower levels of the pattern language help guide the programmer towards an efficient parallel implementation. He or she starts by applying parallel algorithm strategy patterns, which are high level approaches to exploiting concurrency such as *Task Parallelism*, *Data Parallelism*, and *Pipeline*. Below this are the implementation strategy patterns, which are software mechanisms that help the programmer implement efficient parallel code. At the bottom are concurrent execution patterns, which are separated into patterns for advancing program counters, and patterns for communicating between concurrent tasks.

Selective Embedded Just-In-Time Specialization (SEJITS)

As mentioned earlier, patterns are not software artifacts in themselves. Frameworks are a vehicle for encoding pattern knowledge into a usable software artifact. Selective embedded just-in-time specialization (SEJITS) is a methodology for designing frameworks that allow for runtime code generation of selected patterns and computations.

SEJITS was designed explicitly to tackle the implementation gap [9]. Starting with code written in a scripting language, a SEJITS *specializer* will inspect the abstract syntax tree that it is to execute, perform transformations and code generation, JIT compile the code, execute it, and cache it for future use. This process is *selective*, meaning it only takes place on certain portions of the code that are computationally intensive, and it is *embedded* in the productivity language, making it directly accessible to productivity programmers. This provides a natural division of labor between the efficiency programmer, who designs the specializers, and the productivity programmer, who invokes them in his or her applications. It also provides a natural setting for hardware-specific tuning or autotuning. The specializer has access to both the program and the hardware information, at runtime. The SEJITS methodology has been successfully applied in bridging the implementation gap for stencil computations, communication-avoiding algorithms, and audio analytics [74, 75, 13]. The

success of this work also led to the create of the ASP framework, which makes SEJITS infrastructure available for more general use [66].

2.7 Summary

In this chapter, we provided relevant background material. We introduced two important classes of parallel hardware, CPUs and GPUs, and discussed the parallel programming problem. To motivate our framework in Chapter 5, we introduced four framework requirements. We summarized related work and the SEJITS methodology.

Chapter 3

Optimizing for Problem Shape

In Chapters 3 and 4 we focus on optimizing dense linear algebra computations on GPUs using extensive hand tuning. We show that both the shape and the size of the matrices involved in these computations affects which implementation approaches perform best. This motivates our assertion that knowledge of shape and size information is beneficial for achieving high performance implementations.

In this chapter we focus on the QR factorization, a frequently used linear algebra routine that can, among other things, solve least squares problems. For a particular shape of tall-skinny matrices, where the number of rows is much larger than number of columns, we show that a communication-avoiding (CAQR) algorithm is faster than traditional approaches.

We also discuss stationary video background subtraction, a motivating application of the QR factorization. We use a recent statistical approach to stationary video background subtraction, which requires many iterations of computing the singular value decomposition of a tall-skinny matrix. Using CAQR as a first step to getting the singular value decomposition, we are able to get the answer 3x faster than if we use a traditional bandwidth-bound GPU QR factorization tuned specifically for that matrix size, and 30x faster than if we use Intel's Math Kernel Library (MKL) singular value decomposition routine on a multicore CPU.

3.1 Introduction

One of the fundamental problems in linear algebra is the QR decomposition, in which a matrix A is factored into a product of two matrices Q and R, where Q is orthogonal and R is upper triangular. The QR decomposition is most well known as a method for solving linear least squares problems, and is used commonly across all of dense linear algebra.

In terms of getting good performance, a particularly challenging case of the QR decomposition is tall-skinny matrices. Examples of tall-skinny matrices are when the ratio of rows to columns are 3 to 1, 1,000 to 1, or even 100,000 to 1 in some cases. QR decompositions of this shape matrix, more than any other, require a large amount of communication between processors in a parallel setting. This means that most libraries, when faced with this

problem, employ approaches that are bandwidth-bound and cannot fully make use of the floating-point capabilities of processors.

Matrices with these extreme aspect ratios would seem like a rare case; however they actually occur frequently in applications of the QR decomposition. The most common example is linear least squares, which is ubiquitous in nearly all branches of science and engineering and can be solved using QR. Least squares matrices may have thousands of rows representing observations, and only a few tens or hundreds of columns representing the number of parameters. Another example is stationary video background subtraction. This problem can be solved using many QR decompositions of matrices on the order of 100,000 rows by 100 columns [76]. An even more extreme case of tall-skinny matrices are found in s-step Krylov methods [77]. These are methods for solving a linear equation $Ax = b$ by generating an orthogonal basis for the Krylov sequence $\{v, Av, A^2v, ...A^nv\}$ for a starting vector v. In s-step methods, multiple basis vectors are generated at once and can be orthogonalized using a QR factorization. The dimensions of this QR factorization can be millions of rows by less than ten columns.

These applications demand a high-performance QR routine. Extracting the foreground from a 10-second surveillance video, for example, can require over a teraflop of computation [76]. Unfortunately, existing GPU libraries do not provide good performance for these applications. While several parallel implementations of QR factorization for GPUs are currently available [78, 79, 80], they all use generally the same approach, tuned for large square matrices, and thus have up to an order of magnitude performance degradation for tall-skinny problems. The loss in performance is largely due to the communication demands of the tall-skinny case. We aim to supplement the existing libraries with a QR solution that performs well over *all* matrix sizes.

Communication-Avoiding QR (CAQR) is a recent algorithm for solving QR decomposition that is optimal with regard to the amount of communication performed [81]. This means that the algorithm minimizes the amount of data that must be sent between processors in the parallel setting, or alternatively the amount of data transmitted to and from global memory. As a result, the CAQR algorithm is a natural fit for the tall-skinny case where communication is usually the bottleneck.

In this chapter we discuss the implementation and performance of CAQR for a single NVIDIA Fermi graphics processor (GPU). It is a distinguishing characteristic of our work that the entire decomposition is performed on the GPU using compute-bound kernels. Despite their increasing general-purpose capabilities, it is still a very challenging task to map the entirety of this particular algorithm to the GPU. In doing so, however, we are able to leverage the enormous compute capability of GPUs while avoiding potentially costly CPU-GPU transfers in the inner loop of the algorithm. The benefits can most clearly be seen for very skinny matrices where communication demands are large relative to the amount of floating-point work. As a result, we can outperform existing libraries for a large class of tall-skinny matrices. In the more extreme ratios of rows to columns, such as 1 million by 192, we achieved speedups of up to 17x over standard GPU linear algebra libraries. It is important to note that everything we compare to is parallel. The speedups we show are a

result of using the parallel hardware more efficiently.

We do not directly use available Basic Linear Algebra Subroutine (BLAS) libraries as a building block. CAQR requires many hundreds or thousands of small QR decompositions and other small BLAS and LAPACK [54] operations to be performed in parallel. Consequently, we had to do significant low-level tuning of these very small operations to achieve sufficient performance. We will discuss and quantify the effect of specific low-level optimizations that were critical to improving performance. We also highlight new trade-offs the GPU introduces, which differ from previous CAQR work on clusters and multicore architectures [81] [82].

Finally, we discuss the application of CAQR to stationary video background subtraction, which was a motivating application for this work. Stationary video background subtraction can be solved using Robust Principal Component Analysis (PCA), by formulating the problem as a regularized nuclear norm minimization [76]. In the Robust PCA algorithm, the video is transformed into a tall-skinny matrix where each column contains all pixels in a frame, and the number of columns is equal to the number of frames. In an iterative process, the matrix is updated by taking its singular value decomposition (SVD) and thresholding its singular values. The SVD can be solved using the QR decomposition as a first step. We will show that our CAQR implementation gives us a significant runtime improvement for this problem.

Several implementations of CAQR exist, as well as Tall-Skinny QR (TSQR), a building block of CAQR that deals only with extremely tall-skinny matrices. TSQR has been applied in distributed memory machines [81] [83] and grid environments [84] where communication is exceptionally expensive. However, previous work in large-scale distributed environments focuses on different scales and types of problems than ours. More recently, CAQR was also applied to multicore machines [82], and resulted in speedups of up to 12x over Intel's Math Kernel Library (MKL) at the time. Note, however, that implementing CAQR on GPUs is a much different problem than on multi-core and it is likely that both will be needed in future libraries and applications.

The chapter is organized as follows. In Section 3.1 we survey previous algorithms and implementations of the QR decomposition. Next we discuss the high-level problem of mapping CAQR to heterogeneous systems in Section 3.3. Section 3.4 describes the specifics of our GPU implementation and tuning process. Section 3.5 contains our performance compared to currently available software. In Section 3.6 we outline our motivating video processing application and the performance benefits of using CAQR. Section 8 draws conclusions.

3.2 Background on QR Approaches

There are several algorithms to find the QR decomposition of a matrix. For example, one can use Cholesky QR, the Gram-Schmidt process, Givens rotations, or Householder reflectors [85]. Even though it is less numerically stable than Householder reflectors or Givens rotations, if the matrix is well conditioned, then Cholesky QR may be accurate enough, and it may also may be the fastest approach because it uses matrix multiplication. CAQR is a form of

Figure 3.1: Blocked Householder QR

the Householder approach, where the Householder vectors are broken up in such a way that communication is minimized.

Householder QR

Most existing implementations of QR for GPUs have used the Householder approach, as does LAPACK. One favorable property of the Householder algorithm is that it can be organized in such a way that it makes use of BLAS3 (matrix-matrix) operations. Specifically, the trailing matrix updates for several Householder vectors can be delayed and done all at once using matrix-multiply. This allows for higher arithmetic intensity on machines with a memory hierarchy because matrix-multiplication can be blocked for good cache performance. Higher arithmetic intensity then leads to better performance. This is called blocked Householder QR, because it allows the updates to the trailing matrix to be blocked in cache. Several implementations of the blocked Householder approach for GPUs are currently available [80] [79] [86]. These are generally all very fast due to the heavy use of well-optimized matrix-multiply routines [87] [78].

Figure 3.1 shows a sketch of one step in the blocked Householder algorithm. Here, a panel of some width less than the total number of columns is factored using the Householder algorithm with BLAS2 (matrix-vector) operations. Next, a triangular matrix T is formed from the inner products of the columns in the panel. Finally, the trailing submatrix is updated using a matrix-matrix multiply of the panel's Householder vectors, T, and the trailing matrix itself. After the update, the next panel is factored, and so on.

From Figure 3.1 we can intuitively understand a shortcoming of the blocked Householder algorithm. For very wide matrices, a significant portion of the runtime is spent doing matrix-matrix multiply, which is a very efficient use of hardware given the available high-performance dense matrix-multiply routines for GPUs [88]. However, if the matrix is skinny, a greater portion of the runtime is being spent in the BLAS2 panel factorization, which is not an efficent use of the hardware because matrix-vector routines are generally bandwidth-bound.

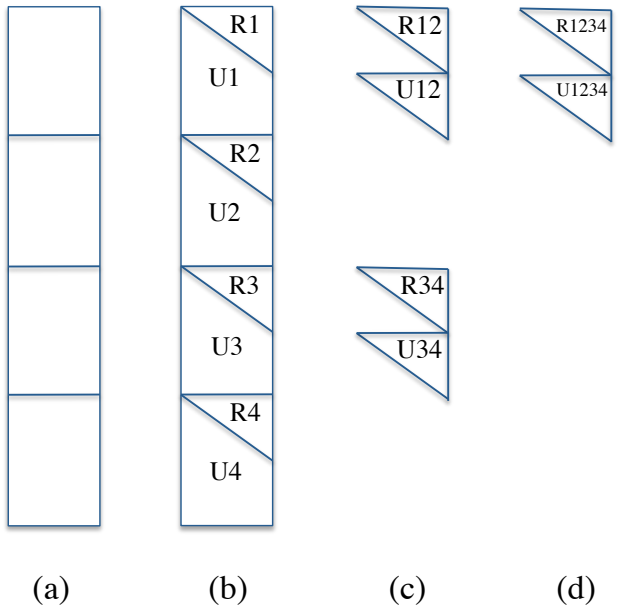


Figure 3.2: Stages of Tall-Skinny QR. First, each block is factored using Householder reflectors. Then, the upper triangular results are stacked and factored until only one upper triangular result remains.

Clever implementations of blocked Householder for heterogeneus CPU+GPU envoronments can hide this cost for wide matrices by sending the panel factorization to the CPU and overlapping it with the previous trailing matrix update, which is performed on the GPU [78]. While this greatly improves the performance for wide matrices, it does not eliminate the latency problem for the skinny case.

Tall-Skinny QR (TSQR)

The TSQR algorithm reorganizes the factorization of a tall-skinny matrix (such as a column panel) to minimize memory accesses [81]. Instead of computing one Householder vector for each column directly, we divide the tall-skinny matrix vertically into small blocks, as shown in Figure 3.2(a). Next, we factor each block independently using Householder reflectors. This creates a small Householder representation of Q, which we call U, and an upper-triangular R for each block. This is shown in Figure 3.2(b). We would like to eliminate all the Rs below the top-most diagonal, so we can group sets together in a stack and apply the Householder algorithm to each (possibly exploiting the sparsity pattern), which is done in Figure 3.2(c). We can continue to reduce the Rs with another level as in Figure 3.2(d). This leaves us with our final upper triangular matrix R, and a series of small Us which, if needed, can be used to generate the explicit orthogonal matrix Q of the factorization.

The TSQR algorithm exposes parallelism. Each block in the panel can be processed independently by a different processor. TSQR also allows us to divide the problem into chunks with a more managable size. If we choose block sizes that fit in cache, we can achieve significant bandwidth savings.

In the figure, the Rs were eliminated in a binary tree. However, this can be done using any tree shape. The optimal shape can differ depending on the characteristics of the architecture. For example, on multi-core machines a binomial tree reduction was used [82], whereas our GPU approach employs a oct-tree reduction. The type of reduction changes how the Q matrix is implicitly represented in memory. That is, the implicit representation of a Q matrix that was generated using a binary tree reduction is incompatable with a routine that uses a oct-tree reduction. The motivation behind our choice to use an oct-tree reduction will be explained in Section 3.4.

Communication-Avoiding QR (CAQR)

CAQR is an extension of TSQR for arbitrarily sized matrices [81]. This time we divide the matrix into a grid of small blocks. Like blocked Householder, CAQR involves a panel factorization and a trailing matrix update. The panel factorization is done using TSQR, shown in Figure 3.3(a). We must then do the trailing matrix update, which means applying the Q^T of the panel to the trailing matrix. Note that because TSQR works on blocks in the column panel, the trailing matrix update can begin before the entire panel is factored. This removes the synchronization in the standard blocked Householder approach and exposes more parallelism. We do not just use a large matrix-matrix multiply as we did in blocked Householder. This is due to the distributed format in which TSQR produces its Q. Instead, we carry out small updates in each small block of the trailing submatrix. Recent work by Ballard et al uncovered a method to recover the Householder vectors that would have been generated with the traditional blocked Householder approach (ie $I - Y * T * Y^T$) [89]. This could be more efficient than the trailing matrix updates described here because it would use matrix-multiplication for the bulk of the computation. We do not attempt this technique in this chapter.

There are two types of trailing matrix updates: horizontal updates and tree updates. Horizontal updates are the easier case. Here we take the Householder vectors generated from the first stage of TSQR and apply them horizontally across the matrix, shown in Figure 3.3(b). This operation is very uniform, and the update of each block in the trailing matrix is independent. The more challenging update is the tree update. Here we take the Householder vectors generated during each level of TSQR's tree reduction and apply them across the matrix. This involves mixing small pieces of the trailing matrix, as shown in Figure 3.3(c) and (d). The rows of the trailing matrix that get updated vary with each level of the reduction tree. This can be challenging on the GPU because the accesses to the matrix are more irregular and somewhat sparse.

After the trailing matrix is updated we can move to the next panel. We must take care to shift the bounds of the working submatrix lower by a number of rows equal to the panel

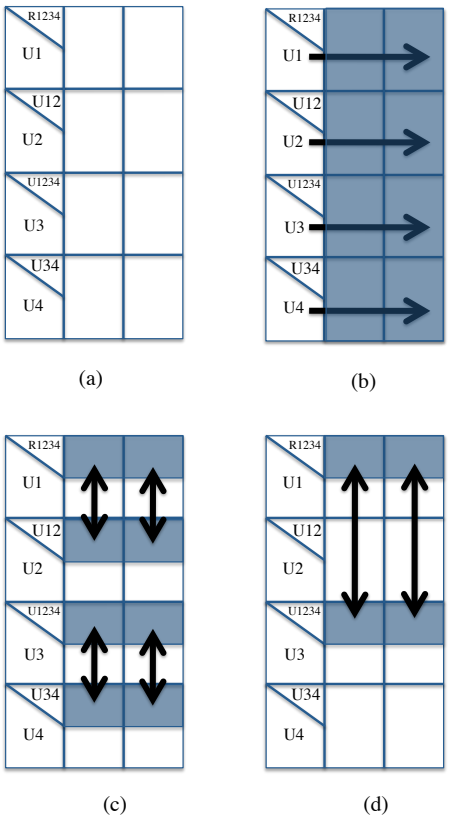


Figure 3.3: Communication-Avoiding QR

width, reflecting the fact that the trailing matrix becomes both shorter and narrower after each step.

3.3 Mapping CAQR to Heterogeneous (CPU+GPU) Systems

Here we briefly discuss two different options for mapping CAQR to current heterogeneous systems. We consider a heterogeneous system containing one or more multi-core host CPUs with DRAM, a GPU with DRAM, and a physical link between the two memories. The GPU has more compute and bandwidth capability generally than the CPUs, whereas CPUs generally have a larger cache, more ability to exploit instruction-level parallelism, and are better equipped to handle irregular computation and data accesses.

The two questions we want to answer with regard to CAQR are: where we should do each step of the computation? Where should we store the data?

First option: CPU panel factorization and GPU trailing matrix update

With this approach, the algorithm proceeds as follows. A panel (a handful of columns) is sent to the CPU, if necessary, and the CPU factors the panel using TSQR. The result of the factorization is sent back to the GPU and used for the trailing matrix update. Potentially, the CPU could begin factoring the next panel while the GPU is still busy applying the previous panel to the trailing matrix.

The main advantage of this approach is that offloading work to the CPU makes it possible to overlap GPU and CPU work. This allows one to use the entire system. The TSQR panel factorization can be a good fit for the CPU because of the irregular nature of the reduction tree. The trailing matrix update is regular and can be done efficiently on the GPU.

One disadvantage of this approach is that in order to offload work to the CPU we must transfer the data between CPU and GPU memories. On current systems, this involves latency that can hurt performance for skinny problems. Unless we successfully overlap CPU and GPU computation, sending the panel factorization to the CPU means we cannot use the superior compute and bandwidth capabilities of the GPU for these parts of the computation.

Second Option: Entire factorization the GPU

With this approach, the entire factorization is done on the GPU. This includes the TSQR panel factorization and the trailing matrix updates.

Assuming the matrix is entirely in GPU memory, this approach eliminates transfer latency. This means that we can get good performance even on skinny problems. We also can benefit from the higher compute and bandwidth capability of the GPU for the panel factorizations.

Unfortunately, this approach is much more difficult to program. This is first because we cannot reuse existing tuned CPU libraries. Also, certain parts of the QR algorithm involve more irregular computations and are therefore more challenging and less efficient to carry out in the GPU's programming and execution models. The pseudocode in Figure 3.4, described in the next section, illustrates some of the irregular operations necessary for a GPU-only approach.

In this work we choose the second option, performing the entire factorization on the GPU, for the following reason. Our motivating application is Robust PCA for stationary video background subtraction. The dimensions of the video matrices we deal with in this application are on the order of 100,000 tall by 100 wide. For this size problem, the latency of transferring data to the CPU will have high adverse impact on performance. During the course of the application, we are doing many QR decompositions and the video matrix is able to stay on the GPU. So the cost of initially transferring the video matrix to GPU memory is easily amortized.

3.4 GPU Implementation

In this section we describe implementation details of our QR decomposition for the GPU. We start with an overview of the hardware, then give a high-level description of how the computation is organized. This is followed by an examination of the key individual kernels and low-level tradeoffs. Everything here is done using single-precision, which is adequate for our video application.

Hardware Overview

Our target is an NVIDIA C2050 GPU, which is the version of NVIDIA's recent Fermi architecture intended for high-performance computing. This choice of architecture is motivated by GPUs' track record for good performance on high-throughput numerical computing [88, 90]. Though we use the C2050, our code can be run on any CUDA-capable GPU.

The 1.15 GHz C2050 has 14 multiprocessors, each of which has 32 floating-point units capable of executing one single-precision FLOP per cycle. In total, the chip is capable of 1.3 single precision TFLOPs.

Tasks are scheduled on the multiprocessors in units called *thread blocks*. Each thread block contains up to 512 threads, which can synchronize and share data among one another through a small shared memory. Thread blocks, however, are assumed to be independent and do not generally synchronize or communicate during a single parallel task. As an example, in our application each small block QR decomposition is done by a different thread block. Within the thread block there are 64 threads that work together to compute the QR.

There is one large global memory (DRAM) shared by all multiprocessors. The peak bandwidth to and from global memory for the C2050 with ECC enabled is 144 GB/sec. There is a small 768 KB L2 cache shared by all multiprocessors. Each multiprocessor has access to 48 KB of shared memory and 16 KB of L1 cache, which can alternatively be configured as 48 KB of L1 and 16 KB of shared memory. The register file is the largest and fastest local memory available to the processing units. It is 128 KB per multiprocessor, can be accessed in one or two cycles, and generally provides enough bandwidth to saturate the floating point units. Registers are not shared. Any time threads need to communicate, in a parallel reduction for example, either shared memory or global memory must be used.

Top-Level Pseudocode

Figure 3.4 shows pseudocode for our CAQR. Each function represents a GPU *kernel*, which is a subroutine that is performed in parallel over different parts of the matrix. Next to each kernel call there is a graphic of our matrix showing how it is divided among thread blocks and which portions of the matrix are being processed by that kernel. So for example, the TSQR panel factorization is done in the first two kernel calls to *factor* and *factor_tree*. Once that has completed, the trailing matrix is updated using *apply_qt_h* and *apply_qt_tree*.

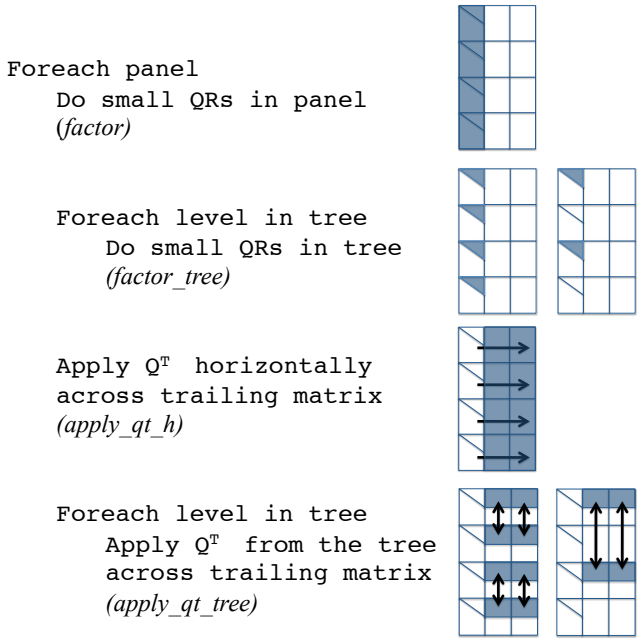


Figure 3.4: Host pseudocode for CAQR with a binomial reduction tree

This pseudocode is executed on the host CPU. The CPU coordinates the GPU's actions and updates pointers so that GPU thread blocks know where to look in the matrix.

Reduction Tree

The shape of our reduction tree is a function of the block sizes. For example, if the block size is 128×16, each block produces an R which fits in a 16×16 space. When these Rs are stacked for the next level of the reduction tree, we can fit $\frac{128}{16} = 8$ of them in each 128×16 block. This means we reduce the height of the panel by a factor of 8 at each level and the reduction is a oct-tree. The tree reduction ends when the panel height becomes less than 128, meaning it can be processed completely in a single thread block.

Overview of Kernels

The following four kernels are the building blocks of our CAQR. The parts of the matrix affected by each kernel are shown in Figure 3.4.

factor

Perform a QR decomposition of a small block in fast memory using customized BLAS2 routines. Overwrite the Householder vectors and upper triangular R on top of the original

small input matrix. While computing the Householder vectors, we do not perform the numerical checks found in the LAPACK SLARFG routine or in the computation of the vector norm. It is likely that doing these checks would reduce performance to some degree.

factor_tree

Gather a stack of upper triangular Rs generated by *factor* and store them in fast memory. Then perform a QR decomposition on that small block, as was done in *factor*. The shape of the resulting Householder vectors and R is also a stack of upper triangular matrices, and thus can overwrite the Rs that were read into fast memory. We do not exploit the sparsity of this stack of upper triangular matrices to reduce the amount of work. The computation proceeds as though it were a dense matrix.

apply_qt_h

Apply Q^T from the Householder vectors generated in *factor* horizontally to small blocks across the trailing matrix. Write back the updated trailing matrix blocks to the locations from which they were read.

apply_qt_tree

Apply Q^T from the Householder vectors generated by *factor_tree* during the TSQR reduction tree to the correct locations in the trailing matrix. To do so, collect the distributed components of the trailing matrix to be updated as well as the distributed Householder vectors. Perform the application of Q^T, and write back the updated trailing matrix blocks to the same distributed locations from which they were read.

Inserting these kernels into the pseudocode in Figure 3.4 should complete a high-level understanding of our CAQR implementation.

Kernel Tuning

Now we examine an individual kernel so as to understand the specifics of the data layout and thread-level execution. Fortunately, all four kernels do the same two core computations: matrix-vector multiply and rank-1 update. We will analyze only the simplest of these which is *apply_qt_h*.

Suppose we have a set of Householder vectors in shared memory and a small block A to which we'd like to apply the vectors. This is equivalent to applying Q^T, represented by the Householder vectors, i.e. $Q = \prod_{i=1}^{n}(I - \tau_i u_i u_i^T); \tau_i = \frac{2}{||u_i||_2^2}$, to the block A. For each Householder vector u we first compute the matrix-vector product $A^T * u$. This is shown in Figure 3.5(a). Then we update each element of A with the scaled outer product of $A^T * u$ and u, Figure 3.5 (b). So the computation here consists of a reduction sum of each column of A during the matrix-vector product, and a data-parallel rank-1 update of A. This is repeated for each Householder vector.

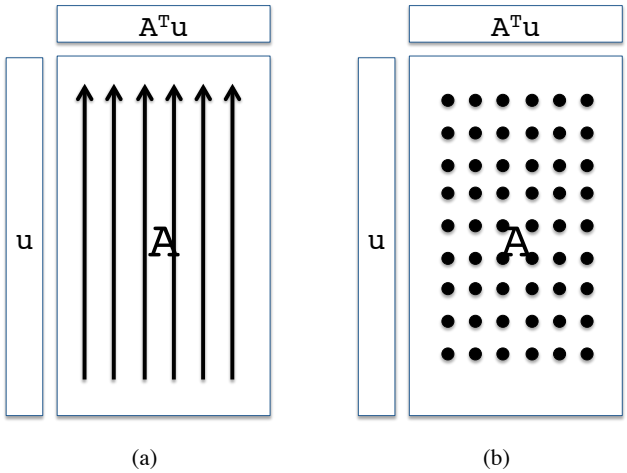


Figure 3.5: Matrix-vector multiply and rank-1 update. These two operations make up the core computations in each kernel

Two important questions must be answered in our design. How should we assign threads to computation, and how do we store the matrix in our fast memories such that it can be accessed most quickly? During the course of tuning we tried four different approaches. The main difference between these lies in how the reductions in the matrix-vector product are carried out. The following is a description of each approach to the matrix-vector product, with its corresponding performance for the matrix-vector product and rank-1 update on 128×16 blocks, arranged in order of increasing performance. .

Approach 1: Shared Memory Parallel Reductions (55 GFLOPS)

The most obvious approach is to store the matrix in the register file assigning each thread one row of the matrix A. Then reduce each column one at a time using parallel reductions in shared memory. This approach is inefficient because many of the threads sit idle during the consecutive parallel reductions.

Approach 2: Shared Memory Serial Reductions (168 GFLOPS)

The 128×16 matrix is originally stored in column major order, so we put it into shared memory to reduce it serially. The advantage of this approach is that it gives us near optimal thread utilization for the reduction.

Approach 3: Register File Serial Reductions (194 GFLOPS)

Store the matrix entirely in the register file. Instead of distributing the data to each thread by row, distribute it cyclically among the threads, as shown in Figure 3.6. Notice that each

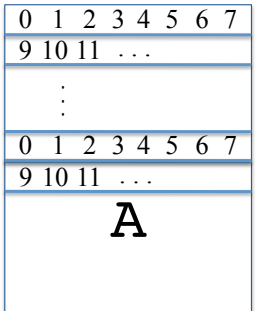


Figure 3.6: In approaches 3 and 4, the matrix is stored in the register file and distributed among threads in this manner. All data owned by a thread belongs to the same column.

thread's data is located in a single column. This means that each thread can do a serial reduction over its part of the matrix and write only the result into shared memory for parallel reduction. This minimizes traffic in and out of shared memory, which helps performance.

Approach 4: Register File Serial Reductions + Transpose (388 GFLOPS)

The register file serial reduction (Approach 3) can be improved if the block is already stored in transposed form. Instead of doing a small transpose in each thread block, this transpose can be done as a preprocessing step. This is beneficial because these kernels are called many times on the same block of the matrix. This is not a transpose of the entire matrix. We only need to transpose each panel from column major to row major. Unfortunately this means that the factorization is done out of place, as an in-place transpose is difficult for non-square matrices.

Tuning Block Size

In the previous section we showed how changing the data layout and reduction technique could improve performance. After committing to a data layout, we can write scripts to test many different block sizes and choose the best.

Our block size is fundamentally limited by our shared memory size and/or register file size. We must however decide what the shape of the block should be. The *apply_qt_h* kernel gets better performance when the block width is wider. This is for several reasons. First, since the number of FLOPs performed in the Householder algorithm is $\mathcal{O}(mn^2)$, we get far higher arithmetic intensity, i.e. FLOPs/byte, by increasing the width n, than we do by increasing the height m. Secondly, the wider the block is the more parallelism there is in each matrix-vector product. If the block width were equal to the number of threads, for example,

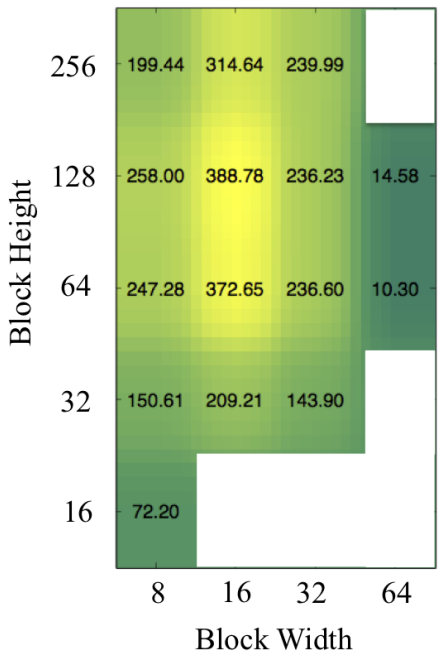


Figure 3.7: Performance for various choices of block size in single precision GFLOPS.

then the matrix-vector product could be done entirely using efficient serial reductions within each parallel thread.

However, the Householder vector u must be communicated to every thread. If the block is so wide that each thread owns an entire column, then each thread must read the entire u vector from shared memory. The optimal solution is somewhere between the two extremes. The performance of each block size is shown in Figure 3.7. Our best overall performance comes from using 128×16 blocks. For the *apply_qt_h* kernel we are able to get 388 GFLOPS. All the performance results for the remainder of the chapter use this block size (128×16) regardless of the size of the problem being solved. This choice of block size also means that the CAQR reduction tree is an oct-tree. The 128×16 block produces a 16×16 upper triangular R matrix, which can be arranged in a stack of 8 and factored again using the 128×16 routine.

Tuning Summary

Through our tuning process we were able to improve the performance of *apply_qt_h*, our main kernel, from 55 GFLOPS to 388 GFLOPS using low-level tuning. The main focus was optimizing the matrix-vector product and rank-1 update that is at the core of the Householder QR algorithm. The most important tuning optimization was to avoid shared memory and L1 cache in favor of the register file. Also, choosing the right block size was critical in achieving good performance.

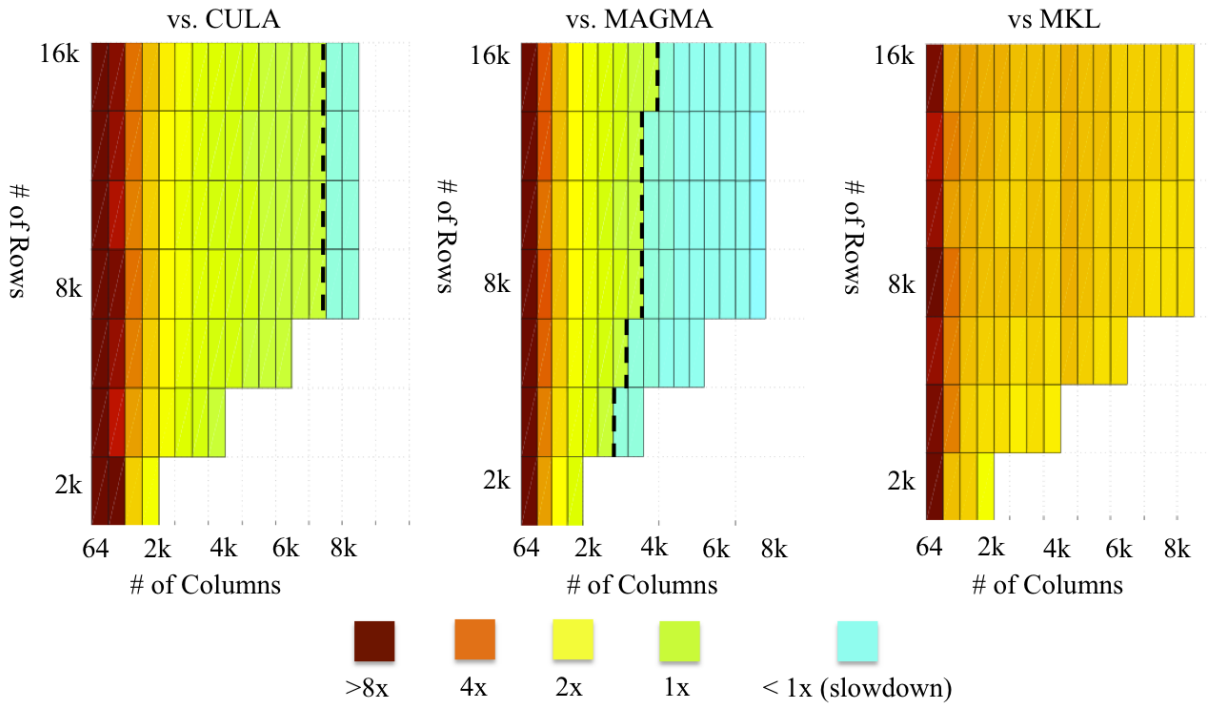


Figure 3.8: Speedup vs. SGEQRF from popular linear algebra libraries on a range of different sizes. The dashed line is a crossover point, to the right of which the libraries outperform our QR. We use a block size of 128x16 for all matrix sizes.

3.5 Performance

We compare our performance against common commercially available and open source software for both GPU and multicore. Below we describe the software to which we compare and the details of the hardware platforms. Our code is optimized for tall-skinny matrices, so we will focus mostly on this case. All FLOPs are single precision.

Software

Intel Math Kernel Library (MKL)

MKL contains BLAS, LAPACK, and various other common functions tuned specifically for Intel's processors. In our comparison we use version 10.2.2.025, and link to the multithreaded library.

MAGMA

Matrix Algebra on GPU and Multicore Architectures (MAGMA) is an open source dense linear algebra library for heterogeneous CPU+GPU systems [91]. It is developed by the

Innovative Computing Laboratory at the University of Tennessee. Version 1.0 from October 2010 is used in this comparison. The current version of MAGMA as of 2014 is 1.5. Since 1.0, MAGMA has added support for newer NVIDIA GPUs, OpenCL, Intel MIC, multiple GPUs and Multicore, and tile LU, QR, and Cholesky factorizations with StarPU dynamic scheduling.

CULA

CULA is a library of LAPACK routines for GPUs [80]. It is a commercial library, made by EM Photonics. Results from October 2010 are used in this comparison. Since CULA source is not available, we do not know exactly how they implement their QR. However, the performance of the QR routine across different square matrix sizes is very similar to the performance of a previous blocked Householder approach by Volkov et al.[78]. For this reason, we will not separately report the performance of CULA and Volkov.

Hardware

Our test platform is the Dirac GPU cluster at the National Energy Research Scientific Computing Center (NERSC) [92]. One node consists of dual-socket quad-core Intel 5530 processors, 8 cores total, running at 2.4 GHz connected over PCI-express to an NVIDIA C2050 (Fermi) GPU. The GPU has ECC enabled, so its effective bandwidth is reduced to 144 GB/s.

Performance vs. Matrix Width

Figure 3.8 shows the performance of our single-precision CAQR code for a range of matrix sizes compared to MAGMA, CULA, and MKL. Each point in the chart represents a different matrix size. The points on the left are skinnier matrices, such as those found in robust background subtraction, and the points on the far right are square matrices. Our CAQR implementation is tuned for the tall-skinny matrices. As a result, we see large speedups compared to other linear algebra libraries for this case. The operation being performed is a single precision QR factorization defined by the LAPACK SGEQRF routine. This routine doesn't return Q explicitly; instead it returns an intermediate representation that can later be used to retrieve or apply Q. Though not shown on the graph, retrieving Q explicitly (SORGQR) using CAQR is just as efficient as factoring the matrix. All preprocessing (e.g. transpose) done for CAQR is included in these runtimes. For all GPU routines, the matrix is assumed to begin on the GPU and therefore the initial data transfer from CPU to GPU is not counted.

The absolute performance numbers in single precision GFLOPS are shown in Figure 3.9. In this case we consider a matrix with a fixed height of 8192 and varying width from 64 to 8192. The crossover point, where CAQR becomes slower than the best GPU libraries, is

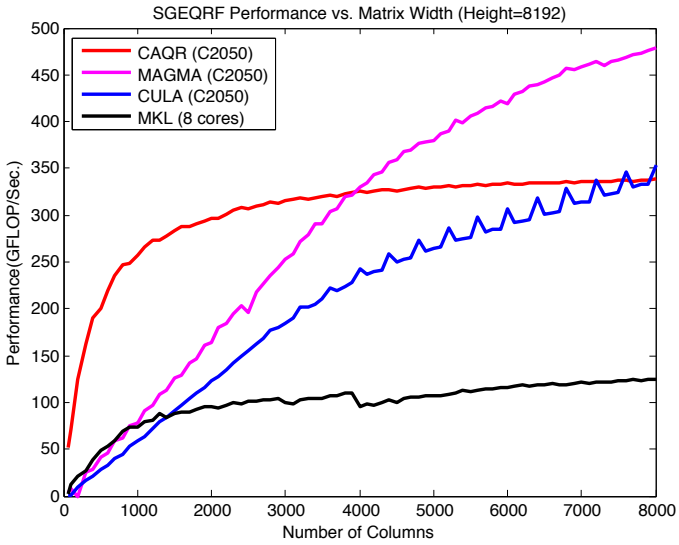


Figure 3.9: Performance on matrices with 8192 rows and varying numbers of columns. For the tall-skinny case CAQR performs best.

		Performance (SP FLOPS)			
		CAQR	MAGMA	CULA	MKL
Matrix Size	1k x 192	39.6	5.01	2.99	3.12
	10k x 192	111	18.7	9.67	16.9
	50k x 192	174	20.8	9.42	22.8
	100k x 192	180	18.8	8.90	21.4
	500k x 192	194	12.4	8.40	17.8
	1M x 192	195	11.4	7.79	16.5

Table 3.1: Performance in single precision GFLOPS for very tall-skinny matrices. We use a block size of 128x16.

around 4000 columns wide. This suggests an autotuning framework for QR where a different algorithm may be chosen depending on the matrix size.

Very Skinny Matrices

In the case of extremely tall-skinny matrices, such as those found in our video processing application, we see up to 17x speedups vs. GPU libraries and 12x vs. MKL. Table 3.1 shows the performance on extremely tall-skinny matrices for CAQR, MAGMA, CULA, and MKL. Another application where matrices like these appear is communication-avoiding linear solvers, when vectors must be orthoganalized periodically [77].

Figure 3.10: Sample output of Robust PCA for stationary video background subtraction. Video is from the ViSOR surveillance video database [93].

3.6 Application: Robust PCA for Surveillance Video Background Subtraction

The motivating application for this work is stationary video background subtraction using a recent statistical algorithm for Robust Principal Component Analysis (PCA) [76]. This section will present specifics of the application, how it uses the QR decomposition, and the performance of the application using ours and other QR implementations.

Robust PCA

Principal Component Analysis is a widely used method for data analysis. The goal is to find the best low rank approximation of a given matrix, as judged by minimization of the difference between the original matrix and the low rank approximation. However, the classical method is not robust to large errors that are sparsely distributed. In Robust PCA, a matrix M is decomposed as the sum of a low rank component L_0 and a sparse component S_0. S_0 is allowed to have entries that are large in absolute value, as long as they are sparse.

$$M = L_0 + S_0$$

The problem is solved using ℓ_1 regularized nuclear norm minimization. Minimizing the nuclear norm, the sum of the singular values, of L_0 enforces low-rank. Meanwhile, the ℓ_1 norm of S_0 enforces sparsity. Minimizing a weighted combination of these two penalty functions with linear constraints is a convex minimization problem.

Stationary video backgrund subtraction is an application of Robust PCA [76]. A survellience video is transformed into a tall-skinny matrix where each column contains all pixels in a frame, and the number of columns is equal to the number of frames. The low-rank component of this matrix is the background and the sparse component is the people walking in the foreground. To give a better idea of the problem being solved, Figure 3.10 shows a sample of the output of the Robust PCA code.

SVD using QR

The main computation in Robust PCA is a singular value decomposition (SVD) of the tall-skinny video matrix. In the SVD of the video matrix, the top singular values, those that have a strong presence in virtually every frame of the video, are usually associated with the background.

Instead of trying to do a large SVD on the GPU, we use the following well known technique for tall-skinny matrices to reduce the bulk of the work to a QR decomposition. First the matrix on which we are performing the SVD, A, is decomposed into $Q * R$. Then we find the SVD of R, which is cheap because R is an $n \times n$ matrix. The SVD is done on the CPU using Intel's optimized Math Kernel Library (MKL). Next, we can multiply the orthogonal matrices $Q * U$ to get the left singular vectors of A.

$$\begin{aligned} A &= Q * R \\ &= Q * \left(U * \Sigma * V^T\right) \\ &= (Q * U) * \Sigma * V^T \\ &= U' * \Sigma * V^T \end{aligned}$$

Robust PCA Algorithm

The algorithm for Robust PCA tries to minimize the rank of L_0 and enforce sparsity on S_0. It does so with an iterative alternating-directions method [94]. The flowchart for the algorithm is shown in Figure 3.11. The algorithm thresholds (sets to zero) the smallest singular values of L_0 in order to make it low rank. Next, a shrinkage operation (pushing the values of the matrix towards zero) is done on S_0 to enforce sparsity. The vast majority of the runtime is spent in the singular value thresholding operation, specifically the SVD of the L_0 matrix.

Performance using CAQR

We implemented three different versions of Robust PCA for video background subtraction. The first runes exclusively on a CPU (Intel Core i7 2.6 GHz), and relies on multithreaded MKL version 10.2.5.035 for the SVD as well as other basic BLAS routines. The second is done entirely on the GPU (GTX480), except for the small SVD of R which is done on the CPU, and uses our BLAS2 QR decomposition that was specifically designed and tuned for tall-skinny matrices. Finally, we implemented a version which also runs on the GPU and uses our CAQR.

Our benchmark video comes from the ViSOR surveillance video database [93]. We extract 100 frames for processing. Each frame is 288 pixels tall by 384 pixels wide, which is a total of 110,592 pixels per frame. This means the matrix dimensions are 110,592 by 100. The problem technically takes over 500 iterations to converge, however the solution begins to look good earlier than that. The solution quality, and therefore number of iterations

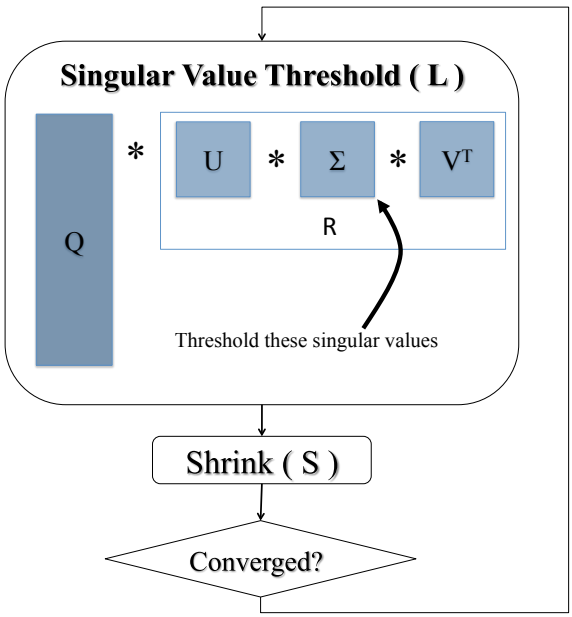


Figure 3.11: Flowchart of the alternating-directions algorithm for solving Robust PCA

required, seems to depend on the application. We therefore report the number of iterations per second that each implementation is able to complete. All computation is done in single precision.

SVD type	Number of Iterations/Sec.
MKL SVD (4 cores)	0.9
BLAS2 QR (GTX480)	8.7
CAQR (GTX480)	27.0

Table 3.2: Performance of various Robust PCA implementations

Table 3.2 shows that moving from the CPU-only code to our BLAS2 GPU code results in a 9.6x speedup. This mostly reflects the fact that the GPU has much higher bandwidth and compute power than our CPU, and that the MKL SVD function may not be optimized for the tall-skinny case. However, we see an additional speedup of about 3x when using CAQR as compared to the BLAS2 QR. Even though the QR itself is sped up by more than a factor of 3, we only get 3x in the application overall due to Ahmdal's law. Overall our GPU solution gives us a 30x speedup over the original CPU code using MKL, reducing the time to solve the problem completely from over nine minutes to 17 seconds, making this approach feasible for latency-critical applications [95].

3.7 Summary

In this chapter we described a high-performance implementation of Communication-Avoiding QR Decomposition entirely on a single GPU using compute-bound kernels. The main advantage of our approach over the traditional blocked Householder algorithm is that it can handle tall-skinny matrices without relying on bandwidth-bound BLAS2 panel factorizations or potentially high-latency GPU-CPU transfers.

We showed low-level implementation choices that allowed us to achieve good performance on the GPU. The best performance for our kernels came from using the register file as much as possible and arranging the data in transposed form so as to minimize necessary communication between threads. Our tuning improved the performance of the most heavily-used kernel from 55 GFLOPS to 337 GFLOPS.

Our CAQR code outperformed leading parallel CPU and GPU libraries for tall-skinny matrices up to roughly 4000 columns wide and 8192 rows tall. In more extreme ratios of rows to columns, such as 1 million by 192, we saw speedups of up to 17x over GPU linear algebra libraries. Note that these extreme cases were motivated by practical applications.

Finally, we applied the CAQR code to Robust PCA for stationary video background subtraction. We showed that using CAQR we could achieve a 3x speedup over our best BLAS2 QR tuned specifically for the tall-skinny case, and a 30x speedup over a CPU implementation using MKL's parallel SVD.

3.8 Conclusion

In conclusion, this chapter showed the benefit both of optimizing specifically for problem shape, and of so called "bare metal tuning." The CAQR algorithm performed best for a certain class of problems, namely those with tall-skinny matrices. But it performed worse than the traditional algorithms for large square matrices. So this tells us that shape information, which may only be known at runtime, is necessary to choose the best implementation. Register blocking and careful selection of block sizes were essential to getting good performance. We conclude that it remains important for efficiency programmers to have access to the "bare metal" in order to achieve high performance.

Chapter 4

Optimizing for Problem Size

In Chapters 3 and 4 we focus on optimizing dense linear algebra computations on GPUs using extensive hand tuning. We show that both the shape and the size of the matrices involved in these computations affects which implementation approaches perform best. This motivates our assertion that knowledge of shape and size information is beneficial for achieving high performance implementations.

In this chapter we focus on solving many (e.g. 2,000) small (e.g. 56×56) linear algebra factorizations in parallel on GPUs. Different parallelization approaches are beneficial for problems this small. We store the entire matrix in the GPU register file and solve it locally, which is much faster than libraries designed for large problems. We design and validate a performance model that predicts the performance of our approaches by counting operations and memory accesses.

4.1 Introduction

Dense linear algebra routines such as solving systems of equations, least squares, and eigenvalue problems are widely used across the computational sciences. Accordingly, it is worthwhile to study these problems closely and tune the algorithms and implementation styles to new architectures.

Several linear algebra libraries are currently available for hybrid CPU+GPU systems that achieve near peak matrix-multiply performance for large, dense factorizations. In this case *large* means thousands of rows and columns. Most current GPU libraries do not efficiently solve small problems, *small* means matrix sizes from 4x4 up to several hundred rows and columns. An exception is CUBLAS version 2, which now includes batched routines. These are not included in this chapter, but we examine them in the case study in Chapter 6.

Small problems do not provide enough work or parallelism to saturate the GPU and they can be solved with relatively low latency on CPUs. However, there are a great number of cases in which many independent small to medium-size problems need to be solved simultaneously. For linear algebra specifically, these problems arise in a diverse range of applications in which

performance matters. One example is MRI reconstruction, which requires solving up to a billion small (8x8 or 32x32) complex eigenvalue problems, one for each voxel in an MRI image, and requires high performance for clinical applicability [96]. Space-time adaptive processing, used in real-time radar signal processing, requires hundreds of simultaneous complex QR factorizations of size 240x66 and is typically limited by the processing capabilities of the radar system [97][98]. To compute observation probabilities with a Gaussian mixture model, large-vocabulary continuous speech recognition applications multiply thousands of 79x16 matrices roughly every one-tenth second [99].

Dealing with many small (e.g. 100x100) matrices on the GPU is different than dealing with large matrices (e.g. several thousands of rows and columns). This is because these two classes of matrices benefit from a different mapping onto both the memory hierarchy and parallelism hierarchy of today's GPUs. Specifically, we distribute independent problems across different multiprocessors and solve them entirely in the multiprocessor register file. In order to understand the design space, we develop a performance model for the GPU that considers both global and intraprocessor communication and we measure relevant parameters with microbenchmarks. This model accurately predicts and explains our performance across different problem sizes.

To summarize, this chapter does the following:

- We present solutions to efficiently solve small QR decomposition, LU decomposition, linear systems, and least-squares problems on GPUs.
- We present an analytical model that accurately predicts GPU performance for these problems by considering both global and intraprocessor communication.

We begin by motivating our GPU performance model and reporting relevant parameters from microbenchmarks in Section 4.2. We describe the algorithmic approaches used for our linear algebra kernels in Section 4.3. We describe our approach for solving problems in the register file and thread block, respectively, in Sections 4.4 and 4.5. We discuss and analyze other existing and potential approaches to solving these problems in Section 4.6. Finally, we present conclusions in Section 8.

4.2 GPU Performance Modeling

We target an NVIDIA Quadro 6000 GPU, which is a GF100 (Fermi) architecture. The relevant specifications of the Quadro 6000 are listed in Table 4.1. The Quadro 6000 GPU has 14 multiprocessors and a total of 448 floating-point units each running at 1.15 GHz. Each CUDA thread is limited to 64 32-bit registers. Each thread block has a 64 KB shared memory unit that can be partitioned between scratchpad and L1 cache. The peak theoretical global memory bandwidth is 144 GB/s and the size of global memory is 6 GB. Floating-point units are capable of doing two single-precision FLOPs per cycle (multiply-add), which adds up to a peak of 1.03 TeraFLOPS for the entire chip.

	Quadro 6000
Number of multiprocessors (SIMT unit)	14
Total number of FPUs	448
Core clock rate	1.15 GHz
Max registers per FPU	64
Shared memory per SIMT unit	64 KB
Global memory bandwidth	144 GB/s
Global memory size	6 GB
Peak SP flops	1.03 TFlop/s
Peak SP per FPU	2.3 GFlop/s

Table 4.1: Summary of the Nvidia GF100 chip and the Quadro 6000.

Table 4.1 shows an overview of relevant performance model parameters for the NVIDIA Quadro 6000 GPU, measured using a set of microbenchmarks [100]. These parameters are used later in the chapter as part of a predictive model for explaining the performance of various small dense linear algebra problems. The measured global memory bandwidth was 108 GB/s, which is less than the peak theoretical global memory bandwidth. It, along with the total bandwidth out of shared memory, is listed as inverse bandwidth in order to be consistent with the performance models used later in the chapter. Global memory latency was measured at 570 cycles using a pointer chasing benchmark. Shared memory latency was measured at 27 cycles using a similar pointer chasing benchmark. Synchronization cost for threads within a thread block varies depending on the number of active threads, but in the case of 64 threads it is 46 cycles. The measured the pipeline latency for floating point operations is 18 cycles.

	Quadro 6000
Global memory latency (α_{gbl})	570 cycles
Global memory inverse bandwidth (β_{gbl})	$\frac{1}{108}$ s/GB
Shared memory latency (α_{sh})	27 cycles
Shared memory inverse bandwidth (β_{sh})	$\frac{1}{880}$ s/GB
Synchronization of 64 threads in a SIMT (α_{sync})	46 cycles
Pipeline latency for FP operations (γ)	18 cycles

Table 4.2: Relevant parameters used in our model of GPU performance [100].

4.3 Linear Algebra Algorithms

We focus on the following basic linear algebra factorizations that solve systems of linear equations, least squares, and are building blocks for many more complex algorithms [85]. All of these factorizations have arithmetic complexity $O(n^3)$ and operate on a matrix that has $O(n^2)$ words. As a result, these algorithms can generally be made compute-bound for most architectures and problem sizes.

Gauss-Jordan Elimination

This algorithm solves a system of equations $Ax = b$ by converting A to reduced row echelon form using row operations and applying the same operations to the vector b producing x. We do not do any pivoting in this implementation. This means that the solver will not always produce the right answer, but it provides an upper bound on performance. The vector b is attached to the right side of the matrix. We proceed from left to right, scaling each row by the diagonal element and updating everything to the right of the current column with an outer product of the scaled row and current column. This algorithm performs n^3 FLOPs where n is the dimension of the matrix.

LU

The LU factorization is a FLOP-efficient method of solving linear systems. Instead of reducing the matrix to reduced row echelon form, LU produces a lower triangular matrix L and upper triangular matrix U such that $A = LU$. The solution to $LUx = b$ can then be solved using forward and backward substitution. Our implementation does not pivot so the output of the factorization is simply the lower triangular L and the upper triangular U written over the original matrix A. Not pivoting means that this solver will not always produce the right answer, but it provides an upper bound on performance. We proceed from left to right, scaling each column by the diagonal element and updating the Schur complement with the outer product of l, the scaled column, and u, the current row. This algorithm performs $\frac{2}{3}n^3$ FLOPs.

QR

The QR factorization decomposes an $m \times n$ matrix A into an orthogonal matrix Q and an upper triangular matrix R. This is a numerically stable way to solve least squares (when $m > n$) and systems of linear equations (when $m = n$). There are several algorithms that can be used to compute the QR factorization of a matrix. For example, one could use any of the following algorithms: Cholesky QR, Gram-Schmidt, Givens rotations, or Householder reflectors. Unfortunately, Cholesky QR and Gram-Schmidt are numerically unstable, so we are limited to using either Givens rotations or Householder reflectors. We use the Householder algorithm because it is consistent with the approach that the LAPACK

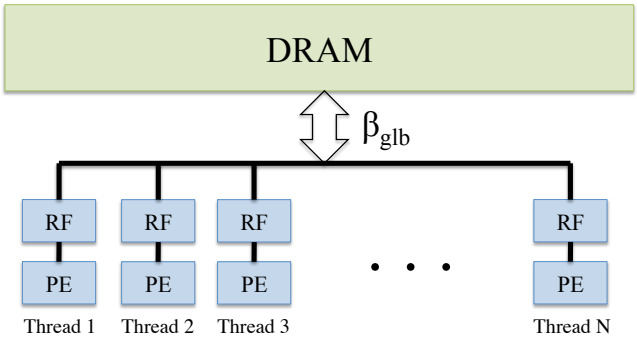


Figure 4.1: Simplified GPU model for the one-problem-per-thread approach. The register files (RF) of each thread's processing element (PE) are connected to a global bus which is capable of moving data at the speed of global bandwidth (β_{glb}). Once in the register file, all FLOPs performed on the data are considered to be free.

library uses. We use an un-blocked (BLAS2) implementation [85]. The Householder QR algorithm performs $2mn^2 - \frac{2}{3}n^3$ FLOPs.

Least Squares

Least squares can be solved using QR by rewriting the normal equations ($A^TAx = A^Tb$) in terms of Q and R ($(QR)^T(QR)x = (QR)^Tb$) that simplify to $Rx = Q^Tb$. So we simply have to find the QR factorization of A, apply Q^T to b, and solve the resulting upper triangular system for x. Note that this is more numerically stable than solving the normal equations directly. We compute Q^Tb by appending b to the right side of the matrix during the factorization. We then solve the upper triangular system using row operations. The total number of FLOPs done in our least squares solver is $2mn^2 - \frac{2}{3}n^3 + \frac{1}{3}n^3$.

4.4 One Problem Per Thread

For very small problems (e.g. $n < 8$) it is possible for each thread to store most of the matrix in its register file and solve the problem serially. Each thread works independently and there is no communication between threads. We choose to register allocate the matrix rather than relying on the L1 cache or shared memory because the register file is significantly faster and over twice the size of these other memories. Register array indices must be known at compile time, so we unroll loops using `#pragma unroll` and C++ templates. Problem sizes that exceed the maximum number of registers available per thread automatically spill into the L1 cache and eventually into DRAM. On GF100, the maximum number of registers per thread is limited to 64. We use the compiler flag `--use_fast_math` that allows for the use of hardware reciprocal and square root functions. These hardware-accelerated functions

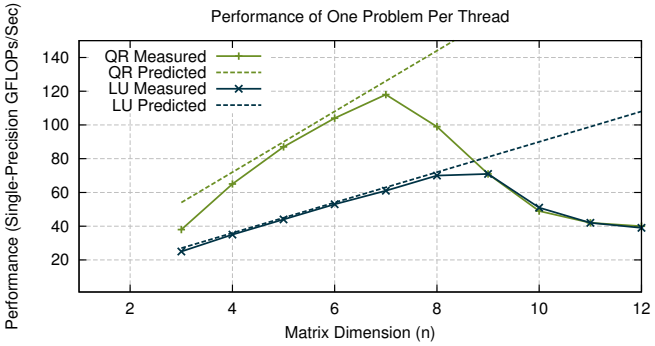


Figure 4.2: Performance for 64000 independent small QR and LU (No pivoting) factorizations. The dashed lines indicate model-predicted performance. For dimensions past 8 the problems no longer fit in the register file and the problems run at the speed of DRAM.

are approximations, but they are accurate up to 22 mantissa bits [18]. In this approach, the median performance penalty for not using these hardware functions is 5.6%.

To understand the performance of this approach we consider the simplified model pictured in Figure 4.1 and described in Section 4.2. We assume that FLOPs have zero latency ($\gamma = 0$) and the register file is infinite. We only count the bandwidth cost between DRAM and register files, as specified by the global DRAM bandwidth (β_{glb}). We also choose to ignore global DRAM latency ($\alpha_{glb} = 0$) since we assume this latency is sufficiently hidden through multi-threading. Expected performance is simply the product of the problem's arithmetic intensity and the global DRAM bandwidth [35].

A problem's arithmetic intensity is the total number of FLOPs performed divided by the total number of words moved. For example, a 7×7 single-precision QR factorization performs $2mn^2 - \frac{2}{3}n^3 = 457$ FLOPs. The entire matrix must be read and written, which generates DRAM traffic of $2 \times 7 \times 7 \times 4$ bytes $= 392$ bytes. Thus, the arithmetic intensity of this problem is $\frac{457}{392} = 1.17$ FLOPs/byte. Our measured bandwidth is $\beta_{glb} = 108$ GBytes/s. We can expect 1.17×108 GBytes/s $= 126$ GFLOPS, which roughly matches the measured performance.

Figure 4.2 shows the expected and measured performance of both LU and QR factorizations for problems of size $n = 3$ to 12 with the one-problem-per-thread approach. Performance follows arithmetic intensity nearly perfectly for both LU and QR until $n = 8$, at which point the problem no longer fits in the register file and spills to L1 and DRAM. For problems where the matrix fits in the register file ($n < 8$) we can say that the bandwidth cost of reading in and out the matrix presents an upper bound on performance. This implementation is optimal in that case.

We could extend the one-problem-per-thread approach to larger problems and potentially sustain the same high performance by using blocked algorithms within a thread [101]. However, for a single-level memory hierarchy such as the GPU's, the performance of this approach would even then face theoretical limits determined by the amount of global band-

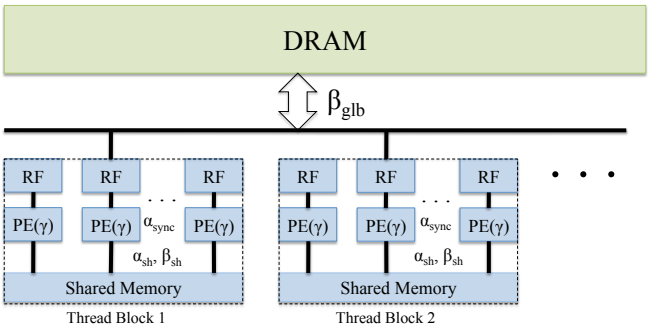


Figure 4.3: Complete GPU model for the one-problem-per-block approach. Each thread block can access DRAM at the speed of global DRAM bandwidth (β_{glb}). Once the data is inside the thread block, it can be stored in the thread's register files (RF) or in shared memory. All threads can access shared memory with a bandwidth (β_{sh}) and latency (α_{sh}). A FLOP takes γ cycles and the operands must be in the register file.

width and the amount of local storage per thread, and regardless of the blocking strategy or algorithm [33]. Another approach to handling larger problems is to assign multiple threads to work together to solve one problem. This approach can increase the amount of local storage per problem and thus the arithmetic intensity. This approach is analyzed next.

4.5 One Problem Per Block

In the previous section we considered the case in which every thread on the GPU loaded a small matrix into its register file and performed a factorization locally. However, if multiple threads work together in a block (e.g. 64 or 256 threads), then we can factor larger matrices without having to communicate with DRAM. One benefit of larger matrices is increased arithmetic intensity. On GF100, 256 threads can store a 112x112 single-precision matrix in a distributed fashion with each thread storing a 7x7 sub-matrix. A QR factorization on a 112x112 matrix performs 1.87 MFLOPs while moving only 100 KBytes of data to and from DRAM. Following the simple bandwidth-only model from Section 4.4 tells us the potential performance of this problem is over 2 TFLOPS, which is beyond the maximum theoretical arithmetic throughput for the chip.

Since the problem is no longer bandwidth-constrained, the costs of doing arithmetic, communicating data between threads in a block, and synchronizing within a block are clearly going to limit performance. These costs are captured by the γ, α_{sh}, β_{sh}, and α_{sync} parameters, respectively. The complete model that is used to understand this approach is drawn in Figure 4.3.

Next, we describe the implementation details of the one-problem-per-block approach. This includes how the matrix is distributed across threads, how threads communicate, and

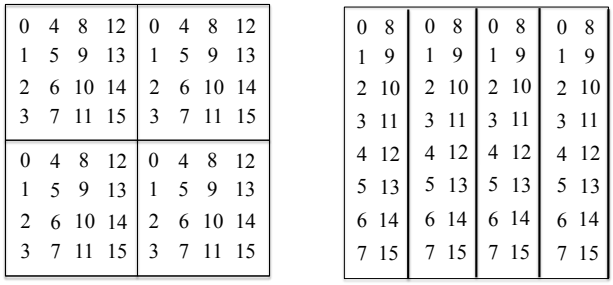


Figure 4.4: A 2D cyclic layout (left) and 1D row cyclic layout (right). Numbers indicate which thread owns each matrix entry.

how computation is divided among threads. Results and analysis with the performance model will follow that.

Distributed Data Layouts

Even though threads have access to a global memory space, the thread block is essentially a distributed system. Each thread represents a machine in the distributed system and each thread's register file represents the private memory. GF100 is a load-store architecture, so all data must be communicated to the register file even for operations with shared memory operands. As in any distributed system, a decision must be made about data layout. We consider three classic distributed data layouts: 1D row cyclic, 1D column cyclic, 2D cyclic. A sketch of these layouts is shown in Figure 4.4.

The simplest data layouts are 1D row or column cyclic. In a 1D row cyclic layout each thread is assigned a row that it stores locally in its register file. If there are more threads than rows, then a row is divided between several threads, preferably in a way that divides the number of elements per thread evenly. 1D column cyclic is similar except columns are assigned instead of rows. The traditional advantages of 1D layouts are that either row or column operations (e.g. computing a Householder reflector) can be carried out within a thread without any communication. One major disadvantage is the load imbalance that occurs in factorizations that proceed from left-to-right, as one thread must drop out after each column is processed.

In a 2D layout, threads own elements from several rows and several columns. We consider a 2D cyclic layout in which elements are distributed evenly throughout the matrix. The 2D cyclic layout mitigates the load imbalance problem inherent in 1D-layout one-sided factorizations, but it introduces communication between $(\sqrt{p})$ threads for both row and column reductions, where p is the number of parallel threads per problem. This can be seen as a compromise between the 1D row and 1D column layouts.

Figure 4.5 shows the performance we achieved using 1D row and column layouts and 2D cyclic layouts on solving systems of equations using a Householder QR factorization followed

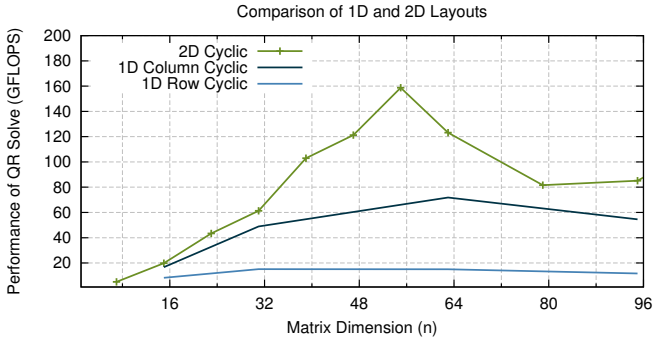


Figure 4.5: Solving 10,000 single-precision linear systems using QR with one-problem-per-block approach and various ways of laying out the data in the register file.

by solving the resulting triangular system using row operations. Due to the large amount of column-wise communication inherent in the Householder QR algorithm, one expects the 1D column-cyclic layout to be considerably faster than the 1D row-cyclic layout. The 2D layout dominates 1D layouts in all tested cases, so we will use this layout by default for the remainder of the chapter. The MAGMA BLAS library for GPUs also uses a 2D register layout for their matrix-matrix multiply routine [102].

Implementation Details

We implemented the factorizations described in Section 4.3 using the one-problem-per-block approach for both square and non-square matrices. The problem dimensions are hardcoded because register file indices must be known at compile time. However, each implementation is parametrized with compile-time constants. This allows the same code to be compiled for problems of several different sizes and shapes. Another way to solve this problem, code generation based on information only known at runtime such as problem size, would be to use selective-embedded just-in-time specialization (SEJITS) [9].

The matrix is stored in the register file in a 2D cyclic layout. The following code loads the matrix from DRAM to the register file. WREG and HREG are the width and height, respectively, of the registered sub-matrix. RDIM is $\sqrt{p}$, where p is the number of threads in a block. The global pointer d_A has already been offset to point to the correct matrix for this block and the correct starting location in the matrix for this thread. This code achieves over 90 GB/sec. for most matrix sizes and thread configurations despite some non-contiguous memory accesses.

To begin an LU factorization, we must modify a column by scaling it, and then copy that column to shared memory. There it can be used to update the trailing matrix. Scaling the vector requires a broadcast of the scaling factor. Listing 4.2, found in LU and Gauss-Jordan, creates a scale factor and assigns it to *scale*. This is a shared memory variable. The variable N indicates the panel being factored (0 through $\frac{n}{\sqrt{p}}$). Variable j is the location of the column

```
#pragma unroll
for(int j = 0 ; j < WREG ; j++)
  for(int i = 0 ; i < HREG ; i++)
    A[i][j] = d_A[i*RDIM + j*RDIM*lda];
```

Listing 4.1: Loading a matrix into the register file

being scaled in the panel. Variable *col* is the thread's column location in the column panel, and *tid* is the thread's row location in the row panel. Alternatively, since threads are laid out in a $\sqrt{p} \times \sqrt{p}$ grid, (*tid*,*col*) are the thread's coordinates in that grid.

```
if(tid == j && col == j) {
  if(A[N][N]!=ZERO){scale = ONE/A[N][N];}
  else {scale = ZERO; *not_solved = 1;}
} __syncthreads();
```

Listing 4.2: The thread on the diagonal determines the scaling factor and assigns it to shared memory.

Scaling and copying the column vector from the matrix in the register file to shared memory is shown in Listing 4.3.

```
if(col == j) {
  #pragma unroll
  for(int ii = N ; ii < HREG ; ii++)
    l[col+ii*RDIM] = A[ii][N] * scale;
}
```

Listing 4.3: Scaling while extracting a column from the matrix in the register file to a vector l in shared memory.

Finally, updating the trailing matrix involves matrix-vector operations such as matrix-vector multiply and rank-1 update. A matrix-vector multiply requires many reductions across threads and is not shown here. Listing 4.4 shows a rank-1 update in which two shared memory vectors l and u are broadcast to update the trailing sub-matrix in A.

The column operation and trailing matrix update are encapsulated in a loop over columns from 0 to RDIM. This loop completes the factorization of an entire panel and updates the trailing matrix. To factor more than one panel, we encode the panel factorization and trailing matrix update as a C++ template and use tail recursion to unroll the entire factorization. Unrolling is necessary because all accesses to the register file must be known at compile time. Register file arrays cannot be indexed by loop indices, for example. N is a template parameter indicating the current panel that begins at zero. The base case for the recursion

```
#pragma unroll
for(int ii = N; ii < HREG; ii++)
  for(int jj = N; jj < WREG; jj++)
    A[ii][jj] -= l[tid+ii*RDIM]
                * u[col+jj*RDIM];
```

Listing 4.4: Rank-1 update of a by shared vectors l and u

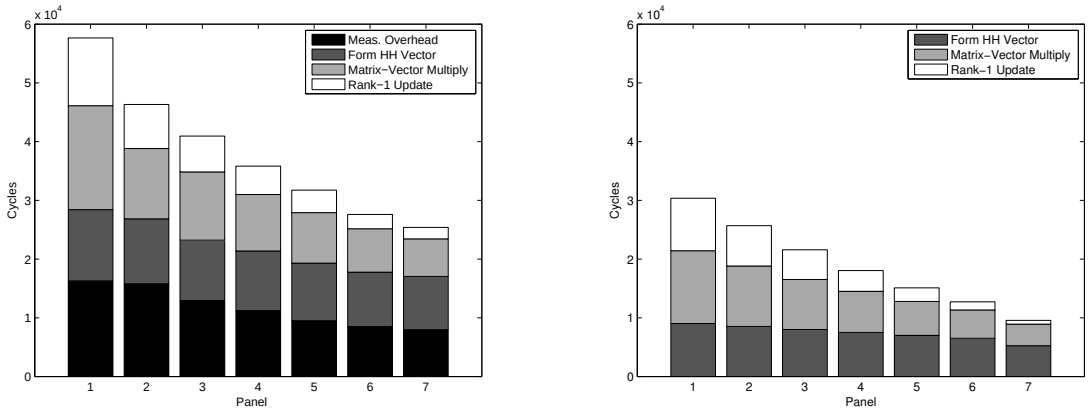


Figure 4.6: The number of cycles spent in each panel of QR measured (left) and modeled (right) for a 56x56 single-precision matrix broken down between the three main operations and measurement overhead [100]. As the factorization proceeds the matrix becomes smaller so each panel takes less time.

is when N = WREG and the factorization is complete. The size of the unrolled code is $O(n^3)$. However there is code reuse equal to the total number of columns in the panel, which is either 8 or 16 in our implementations. The code is also implicitly parallel, so the number of instructions is reduced by a factor equal to the amount of parallelism in each instruction.

Measured Performance

	Load	Compute	Store
LU	8800	68250	8740
QR	9120	150203	9762

Table 4.3: Cycle counts for 56x56 LU and QR decompositions [100].

The average number of cycles measured for a 56x56 single-precision LU and QR factorization for the one-problem-per-block approach is shown in Table 4.3. The cycles are broken

down between memory access time and on-chip compute time. For the 56x56 size, the GPU is executing eight thread blocks per multiprocessor for a total of $14 \times 8 = 112$ problems simultaneously across the entire chip. Each thread block has 64 threads. The measured 9000 cycles for reading and writing seems to indicate an overlapping of global communication with computation, so that fewer than 8 thread blocks are competing for memory bandwidth at a time. If all 8 thread blocks were able to load the matrix in 9000 cycles, this would mean we were achieving nearly double the peak bandwidth that we measured earlier.

Again, we used the compiler flag `--use_fast_math`, which allows for the use of hardware reciprocal and square root functions that are accurate up to 22 mantissa bits [18]. For this approach, not using the hardware functions resulted in a median performance penalty of 30%.

The measured compute times are broken down further in Figure 4.6 into panels and operations within each panel. Each panel has $\sqrt{p}$ columns, so there are 7 panels in a 56x56 matrix with 64 threads. With each new panel the matrix becomes smaller by $\sqrt{p}$ rows and $\sqrt{p}$ columns, so the total time needed to factor that panel and update the trailing matrix decreases over time. Figure 4.7 also contains our model's estimated cycle counts for the same operations. The next section will explain how these estimates were generated.

Modeling Performance

We estimate the performance of LU and QR using the one-problem-per-block approach by counting the number of FLOPs, accesses to shared memory, and thread synchronizations present in our implementation. We assume there is no overlap of either shared or global memory communication and computation. The Gauss-Jordan and least squares solvers are made up of code that is essentially similar to LU and QR. Therefore, we will not analyze them here.

We are not attempting to define upper or lower bounds for the performance on these problems. Rather, we are just trying to understand and predict why our specific implementation runs at the speed it does and where specifically the time goes. There are certainly other ways to solve these problems in the thread block that may perform more or less communication.

The following paragraphs describe how we arrived at the computation and communication estimates for our LU and QR implementations. The final estimates are shown in Table 4.4. For the purposes of this analysis, a floating-point multiply-add is counted as one γ because the pipeline is dual-issue.

This LU implementation can be broken down into two phases: the column operation and the trailing matrix update. Each happens $n-1$ times, where n is the number of columns in the matrix. To compute and communicate the scale factor, the column operation requires a division followed by a shared memory write and a synchronization. Then there is another shared memory read of the scale factor and N FLOPs to scale the column, where N is the height of the current column divided by the $\sqrt{p}$ threads in that column, rounded up if necessary. Finally, the column l and row u are written into shared memory, which requires $2*N$ shared memory accesses and a synchronization at the end. These writes cannot happen

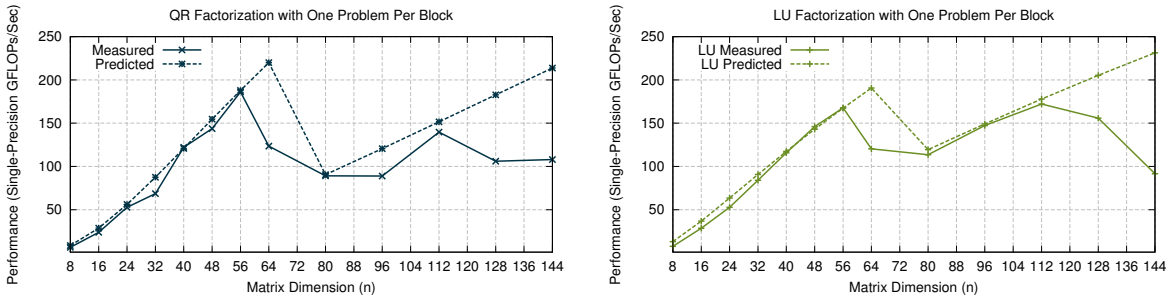


Figure 4.7: The performance of 8,000 LU and QR factorizations with the one-problem-per-block approach. The dashed line indicates the model-predicted performance. The sharp drop from 64 to 80 happens because we switch from 64 to 256 threads. The false predictions at 64 and above 112 are due to register spilling, which our model does not consider.

in parallel because the thread on the diagonal owns some of both l and u. The panel update is a rank-1 update. Each thread must read N elements from both l and u for a total of $2 * N$ shared memory accesses. Then each thread performs N^2 FLOPs and 1 synchronization at the end.

For the QR factorization we choose to do serial reductions instead of parallel. Each reduction is across $\sqrt{p}$ threads so $cost_{red}$ will be $(1 + \sqrt{p})\beta + \sqrt{p}\gamma$. For the matrix-vector multiply we assume that there are at least as many threads as columns so the total cost will be the $cost_{red}$. A column norm reduction will have the same cost as a matrix-vector multiply reduction even though only one thread is needed for the column norm reduction.

The column operation of the QR factorization requires N FLOPs plus a reduction by thread 0 to compute the norm. Then thread 0 computes the scale factor that does a square root, two divides, and two multiplies, followed by one shared memory write of the scale factor. Then the column is scaled and written to shared memory, which requires a read of the scale factor, N FLOPs, and N writes to shared memory followed by synchronization. The trailing matrix update does N shared memory reads to get the Householder vector. Then there is a matrix-vector multiply with is N^2 FLOPs, a synchronization, a reduction, and another synchronization. Next there is a rank-1 update which involves N reads from shared memory of the matrix-vector multiply result, N^2 FLOPs, and finally a synchronization at the end. There are a total of $(n - 1)$ column and trailing matrix operations in QR.

We calculate total cycles by plugging in the parameter values from Section 4.2 and adding the cost of reading and writing the matrix from DRAM. We use the division and square root cycle times from a previous benchmarking paper on the GT200 architecture [103]. To derive global GFLOPS estimates from the number of cycles predicted by the model, we multiply the FLOPs done by a single thread block by the total number of blocks executed simultaneously on the chip. Then we divide that by the number of cycles, and multiply by the number of cycles per second. The number of simultaneous blocks is given by the CUDA occupancy

LU Estimates	
Column	
$\gamma_{div}\alpha_{sync}$	Thread 0 compute scale factor
2β	Write and read scale factor
$N\gamma$	Scale l vector
$2N\beta + \alpha sync$	Write l & u to shared
Trailing Matrix	
$2N\beta$	Read l & u from shared
$N^2\gamma + \alpha_{sync}$	Rank-1 update

QR Estimates	
Column	
$N\gamma$	Column norm
$(1+\sqrt{p})\beta + \sqrt{p}\gamma)$	Thread 0 norm reduction
$\gamma_{sqrt} + 2\gamma_{div} + 2\gamma$	Thread 0 compute scaling factor
2β	Write and read scale factor
$N\gamma + N\beta + \alpha_{sync}$	Column scale & write to shared
Trailing Matrix	
$N\beta$	Read Householder vector
$N^2\gamma$	Matrix-vector multiply
$2\alpha_{sync} + (1+\sqrt{p})\beta + \sqrt{p}\gamma$	Matrix-vector multiply reduction
$N\beta + N^2\gamma + \alpha sync$	Rank-1 update

Table 4.4: Estimates of the FLOPs, shared memory communication, and synchronization done in LU and QR

calculator.

Figure 4.7 shows the performance of LU and QR using the one-problem-per-block approach. The dashed lines indicate the performance predicted by the model including the

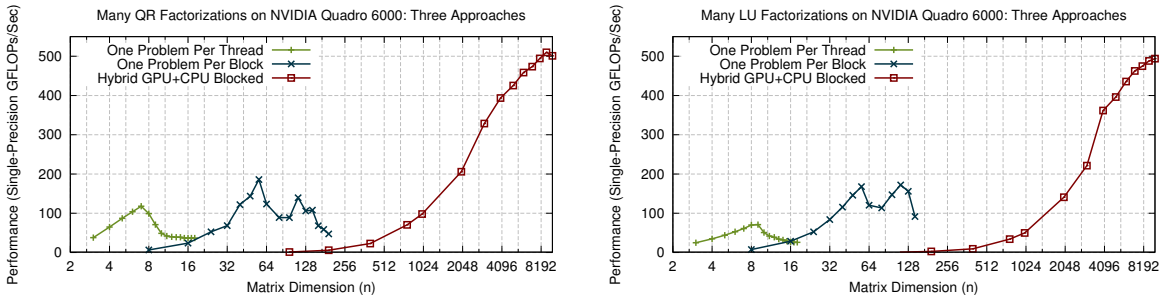


Figure 4.8: The design space for different sized problems is not flat. Our two approaches perform well for thousands of small problems and the MAGMA Hybrid GPU+CPU blocked approach performs well for single large problems.

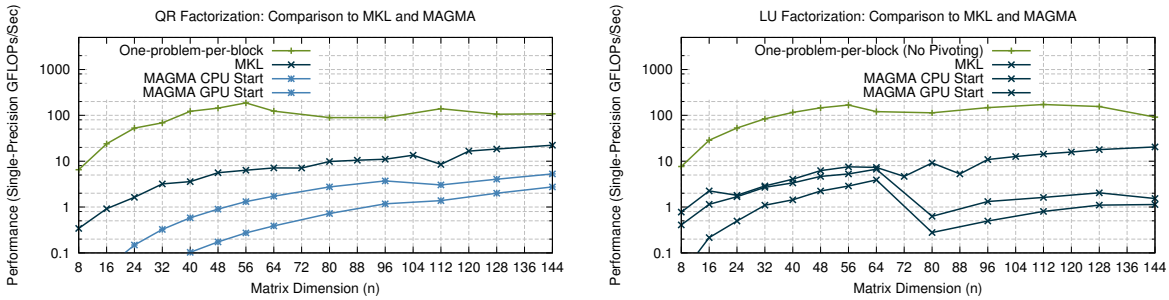


Figure 4.9: Comparison of the one-problem-per-block approach to Intel's MKL and MAGMA GPU linear algebra library for 8,000 LU and QR factorizations. Note our implementation does not pivot during Gaussian Elimination and both MKL and MAGMA do pivot. However, the matrices tested were diagonally dominant so no pivoting was necessary. Doing pivoting using our approach would cause performance to decrease because we wouldn't be able to statically schedule matrix accesses.

bandwidth cost to get the matrix to and from DRAM. Again we can see the effect of register spilling when the number of registers per thread meets or exceeds 64. This spilling is not captured in the model. The abrupt change in performance at $n = 80$ occurs because we switch from using 64 threads per block to 256 threads per block. We switch to 256 threads because the system allows a maximum of 63 register per thread, so in order to handle larger problems we need to increase the number of threads. This reduces the number of simultaneous blocks per multiprocessor from 8 to 2. Note that the number of threads must be a perfect square as a consequence of the 2D layout.

4.6 Other Approaches

Hybrid CPU+GPU Blocked

The predominant approach taken by GPU-enabled linear algebra libraries such as MAGMA and CULA is the hybrid CPU+GPU blocked approach [86][80]. Panels are factored on the CPU and sent to the GPU, where the trailing matrix is updated using matrix-matrix multiply. The trick to this approach is to overlap the communication between CPU and GPU such that the entire problem can run at the speed of the GPU matrix-matrix multiply routine. The other benefit of this approach is that the only GPU code that needs to be written and optimized is the matrix-multiply routine. The downside of this approach is clear for small or skinny problems. In this case the trailing matrix is much smaller, so the majority of the computation, if not all, occurs on the CPU. The panel width in the current MAGMA release is 96 so all problems less than 96 wide are done entirely on the CPU.

Figure 4.8 shows the performance of our approach compared to MAGMA, the state-of-the-art linear algebra library for GPUs. The library does not provide the ability to run multiple problems simultaneously, so we put a loop around the function call and run each problem sequentially. MAGMA provides routines in which data starts and ends on the CPU, and separate routines in which data starts and ends on the GPU. We call the routines in which data starts and ends on the GPU. As this plot illustrates, MAGMA is inefficient for small problems and the overall design space is not flat. For very large problems MAGMA is very fast, for reasons we describe below. However, for small problems our implementation is up to two orders of magnitude faster.

Intel Math Kernel Library (MKL)

Figure 4.9 shows the performance of our one-problem-per-block approach compared to both MAGMA and Intel's Math Kernel Library (MKL) on an Intel Core i7-2600. We distribute the problems evenly across all four cores using pthreads. MAGMA performance is shown for two routines. In the first case the data starts on the CPU. In the second case the data starts on the GPU. The CPU-start is faster because MAGMA solves these problems mostly on the CPU anyway.

These are not ideal comparisons because both MAGMA and MKL do partial-pivoting for LU, while ours does not. However, we ran these examples on diagonally-dominant matrices, so no pivoting should have actually occured. Doing pivoting using our approach would cause performance to decrease because we wouldn't be able to statically schedule matrix accesses. We also use the hardware reciprocal and square root on the GPU, while both MAGMA and MKL use full-precision functions on the CPU.

Using CUBLAS and CUDA Streams

In our implementations we mapped small problems to the GPU memory hierarchy, first to the thread level (one-problem-per-thread), then to the block level (one-problem-per-block). It is also possible to solve these problems at the global level. In this case we can use CUBLAS to do column norms and scales, matrix-vector multiplies, rank-1 updates, and trailing matrix-matrix updates. Since CUBLAS allows for multiple streams we can even call these operations in parallel over multiple problems. However, this approach does not take advantage of the memory hierarchy except for the trailing matrix-matrix multiplies, which are small for these problem sizes. Furthermore, it is practically difficult to get the current GPU to do small CUBLAS routines, such as column norms, in parallel in a fine-grained manner. As a result, our blocked implementation of QR using CUBLAS showed no benefit from using multiple streams. We could achieve better performance solving the problems sequentially on the CPU.

4.7 Summary and Conclusions

We examined the problem of doing many thousands of small dense linear algebra factorizations simultaneously on a GPU. The size of the matrix was very important in determining the best parallelization approach. Very small problems (e.g. $n < 16$) can be efficiently solved by assigning one problem per thread and factoring each problem locally in the register file. This approach is optimal for problems that fit entirely in the register file. For larger problems it makes sense to assign an entire thread block to a single problem because this gives higher arithmetic intensity and requires fewer problems to saturate the GPU. Tiled algorithms can be used to solve problems that are too large to fit in a single thread block's register file.

Our model considers both global (DRAM-to-processor) and local (interprocessor) communication. It is accurate for the LU and QR factorizations we analyzed. These problems have specific qualities that are motivated our specific model. First, we assume that global communication happens separately from local communication. In our case it does because the factorization takes so many more cycles than loading or storing the matrix. Second, there is a significant amount of inter-thread communication that is unlike some previously-studied computationally intense kernels such as matrix-multiply. We believe this model is particularly well suited for other problems that have these two properties.

4.8 Conclusion

In conclusion, this chapter provided another example of the benefit of so-called "bare metal tuning." We also showed that size information could be used to generate tailored implementations for small problems that performed better than libraries that were designed for larger problems. In fact, it was necessary in these implementations to have knowledge of problem size at compile time; in order to use the register file, all register accesses must be explicitly

specified. We used compile-time code generation with C++ templates to solve this problem. In the next chapter, we describe a more general framework that provides the ability to do runtime code generation. The case study in Chapter 6 will revisit the problem of solving small linear algebra problems for a space-time adaptive radar processing application.

Chapter 5

An OpenCL Composition Framework

In order to bridge the implementation gap, we propose an OpenCL composition framework, called *Hindemith*, that uses the selective embedded just-in-time specialization (SEJITS) methodology, along with high-performance hand-coded operations, organized around the principles of pattern-based software engineering. This chapter describes our proposed framework in greater detail. In Section 5.1, we outline the requirements of our framework and how Hindemith addresses these requirements. Section 5.2 describes related work. Sections 5.4 through 5.6 describe how programs are structured and how the efficiency programmer should specify operations. Sections 5.7 and 5.8 describe the fusion and dataflow optimizations that the framework carries out in order to improve performance. Section 5.9 describe how the framework is embedded in the Python productivity language and Section 5.10 summarizes the entire chapter using a Jacobi solver as an example application. The performance of the framework is evaluated later in the case studies in Chapters 6 and 7.

5.1 Requirements

In Section 2.5, we identified four requirements for our framework, which we address here:

- It is transparent and allows for custom hand-tuned code,
- It can take advantage of runtime knowledge such as shapes and sizes of data structures,
- It composes operations for efficiency, taking advantage of data reuse to reduce communication between processors and memory, and
- It is embeddable in a popular productivity language.

The motivation behind each requirement and how we address it with our framework is explained in further detail in the following subsections.

Requirement 1: Transparency

The framework must allow the efficiency programmer to write arbitrary code. This means that the efficiency programmer can be given complete control over the hardware in a low-level language such as OpenCL. Our experiences optimizing dense linear algebra operations in Chapters 3 and 4 demonstrate the benefit of so called "bare metal" tuning.

In this framework, the efficiency programmer can create operations written entirely in OpenCL using any and all of OpenCL's features. These operations can be invoked from Python by the productivity programmer. This works in a similar way to a traditional library. Section 5.6 describes how the efficiency programmer specifies an operation that emits arbitrary OpenCL code.

Requirement 2: Knowledge of Runtime Shapes and Sizes

The framework should allow for runtime code generation with knowledge of problem shapes and sizes. For example, the code for a matrix factorization should be generated and tuned with knowledge of the exact dimensions (width and height) of the matrix. We demonstrate in Chapter 3 that for a QR matrix factorization, different algorithms are more efficient depending on the ratio of matrix height to width (ie tall-skinny problems). In Chapter 4 we demonstrate the need for runtime code generation of small linear algebra factorizations in order to make use of the GPU register file.

In this framework, the datatype, shape, and size of every data structure is propagated through the program at runtime. Each operation, therefore, has access to this information at runtime and can generate different code based on this. Section 5.9 describes the runtime compilation process and the propagation of runtime information.

Requirement 3: Composition

Composition of operations is essential for high performance. This is especially true for light-weight operations such as BLAS1 vector operations or 2D stencils. In order to achieve high performance, the framework must use runtime code generation to fuse related operations into a single loop nest or OpenCL kernel.

Kernel fusion is a feature of several existing GPGPU frameworks. We give an overview of these in Section 5.7. Our framework performs composition of OpenCL kernels at various levels of the OpenCL iteration space. Operations are annotated by the efficiency programmer with dependence information. The framework determines at which level each operation can fuse with every operation and performs the fusion according to a simple (maximal fusion) heuristic. The framework uses dataflow analysis, described in Section 5.8, to improve the efficiency of fused operations by promoting temporary data structures to registers instead of storing them in global memory.

Requirement 4: Embedded in Productivity Language

Python is a popular language among domain experts and productivity programmers. Making a framework programmable from Python gives productivity programmers a reasonable interface to use the framework. It also means that the inputs and outputs of the framework can easily be passed to other libraries, visualization tools, and frameworks that provide Python bindings.

Following the example of the Copperhead framework [12], we embed our framework in Python, meaning all data structures start and end as Python objects, and the framework itself is an external library that processes the program and produces the result. The details of how the framework is embedded in Python are in Section 5.9.

5.2 Related Work

Our work is similar to the Copperhead framework [12, 61]. In Copperhead, the productivity programmer expresses computation as data-parallel operations such as Map, Reduce, and Scan that are built-in to Python. Copperhead intercepts the Python program at runtime, performs inference procedures to determine memory footprint, performs data-parallel primitive fusion, and generates CUDA at runtime. This framework is embedded in Python and operates in the same manner. Our framework differs from Copperhead in that we provide the productivity programmer with arbitrary functions written in OpenCL instead of a fixed set of data-parallel operations. This allows for greater flexibility in hand-tuning individual operations, but also complicates analysis. We also allow operations to have side effects, which Copperhead does not. Finally, our framework has a built-in loop construct and performs dataflow analysis. Copperhead, being a functional language, does not allow for loops and relies on intrinsic properties of functional programming for dataflow information.

PyCASP [13, 63] is a SEJITS framework that lets productivity programmers compose high-performance implementations of content analysis applications, and includes facilities for fusion and runtime code specialization. Our framework differes from PyCASP in that we focus on composing operations in a very fine-grained manner, such as fusing several operations into a single OpenCL kernel and storing intermediate data in registers to reduce communication between processors and memory; PyCASP's fusion occurs at a higher level, avoiding data transfers between CPU and GPU.

There are similar efforts in the image processing domain and the database domain to fuse light-weight operations for efficient parallel execution. Halide [56, 57] is a recent domain-specific language for image processing pipelines that allows the programmer to write transformations on images, along with a specification of the schedule in which the operations execute. Halide composes the image transformations, allows for the duplication of work across parallel elements, and includes an autotuner to search for the best operation ordering. OpenVX [104] is an upcoming standard from Kronos. The OpenVX layer will reside between vision frameworks such as OpenCV, and the OpenCL efficiency language and provide

support for fusion and tiling optimizations. Kernel Weaver [105] is a framework for fusing related light-weight databse operations together into CUDA kernels. This work differs in that we provide a framework for fusion that can be applied to arbitrary OpenCL operations and invoked from Python.

ArrayFire [55] is a parallel programming framework that does just-in-time compilation of array codes to compose operations in such a way as to eliminate unnecessary reads and writes, generating OpenCL at runtime. This work differs from ArrayFire in that we extend analysis and composability to arbitrary OpenCL kernels, as opposed to just linear algebra routines.

PyOpenCL [106] is a tool for executing OpenCL from Python. It provides bindings for OpenCL so that programmers can create OpenCL buffers, create kernels, and execute programs from Python. Our work differs in that we perform analysis and optimization of the OpenCL program.

Finally, this work is a specific instance of a JIT-compiled embedded domain specific language. The Delite project has a similar goal of embedding high-performance parallel operations in a productivity language [64, 65]. The Selective-embedded Just-in-Time Specialization approach been applied to a number of domains including stencils [9], communication-avoiding algorithms [75], dense loop nests [14, 62], and audio analytics [13, 63].

5.3 Jacobi Solver Example

The running example in this chapter is a linear solver, solving for x in the equation $Ax = b$, using the Jacobi method and a matrix represented as a stencil. The solver is written here in Python:

Program 1 Jacobi solver example application written in Python.

```
def jacobi(D, R, b, x):
  for i in range(num_iter):
    r = b - R*x
    x = D * r
  return x
```

The input variable R is the matrix (A) represented as a stencil with the diagonal element removed, x is a two-dimensional array representing an initial guess of the solution, b is the right-hand-side, and D is a scalar containing the inverse of the diagonal element of A. In this particular example, the diagonal is a constant value so we can store it as a scalar. The solver will iterate for *num_iter* iterations and then return the computed solution. If we wanted to terminate early it is possible to check for convergence and add a break statement in the inner loop.

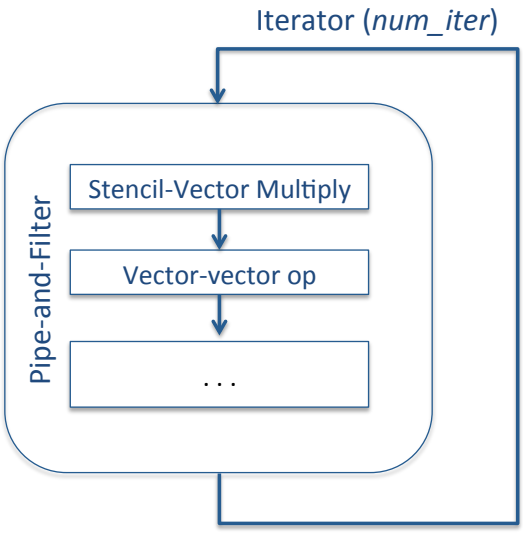


Figure 5.1: Structural patterns for Jacobi solver example.

5.4 Structural Patterns

Applications in our framework may use the either the pipe and filter or iterator structural patterns. These structural patterns were chosen because they are very common; Many applications can be expressed as a hierarchical combination of the iterator pattern and the pipe and filter pattern. These patterns also lend themselves to optimization. In particular, the multiple filters within a pipe and filter can be merged, or composed, for efficiency [13]. This is described further in Section 5.7. The following is a brief description of the supported structural patterns and an example of their use:

- **Pipe and Filter**: Characterized by data flowing through module phases of computation. Filters are units of computation that process data provided through pipes, which are communication channels. Filters are stateless and process data from pipes as it becomes available.
- **Iterator**: Characterized by a body of computation that is repeated until some exit condition is met. The exit condition can be a counter terminating after a fixed number of iterations, or a quality-of-solution metric such as a residual norm that terminates after reaching a certain threshold.

Figure 5.1 shows the structural patterns used for the Jacobi solver example. At the top level there is an Iterator that loops a total of *num_iter* times. Inside the iterator, there is a pipe-and-filter containing the computations from the inner loop of the linear solver. Computational operations fit into the boxes drawn into the figure. Some of these operations are based roughly around computational patterns, and others must be custom

written for each application. For the Jacobi solver, the computation starts with a stencil-vector multiplication, computing $R*x$, followed by a vector operation, subtracting the result from b. The rest of the computation is not shown here.

5.5 Framework Specification

This section provides a specification for programs written in our framework. Programs written in our framework describe the data structures required and the operations being performed on these data structures. The specification is very simple and limited to only structural patterns. However, each operation invoked from these programs can run arbitrary OpenCL.

A program written in the framework consists of two sections:

- **Symbol Table**: Contains declarations of all data structures, including the type (e.g. Array or SpMat), datatype (currently supporting float, double, long, and complex float), and other metadata such as array dimensions.
- **Semantic Model**: Contains a list of operations and loop constructs that specify the behavior of the program.

Symbol Table

The symbol table contains declarations of all data structures and any relevant information such as datatypes or array dimensions. All variables listed in the symbol table have global program scope. Currently supported types are listed here along with brief descriptions:

- **Array**: One-dimensional array. Metadata includes array length and datatype.
- **Array2D**: Two-dimensional array. Metadata includes array height, width, and datatype.
- **Array3D**: Three-dimensional array. Metadata includes array height, width, depth, and datatype.
- **SpMatDIA**: Square sparse matrix in diagonal format [107]. Metadata includes number of diagonals, offsets, and datatype.
- **SpMatCSR**: Sparse matrix in CSR format [36]. Metadata includes matrix height, width, number of nonzeros, and datatype.
- **SpMatDIA2D**: Sparse matrix in diagonal format with diagonals represented as 2D arrays. Metadata includes diagonal width, height, and datatype.
- **Stencil**: Two-dimensional stencil operator. Metadata includes three equal-sized arrays: weights, x offsets, and y offsets.

- **Scalar**: Single value with datatype float, double, long, or complex float.
- **ScalarConstant**: Single value with integer datatype that may not be modified. Value can be used during code generation.
- **InterationVar**: Single value with integer datatype. Value is set by the iterator.

For the Jacobi solver example, the symbol table contains the following inputs: D, R, b, x, r, i, and *num_iter*. The symbol table also contains intermediate data structures that are introduced during the Python compilation process described in Section 5.9.

The symbol table is serialized into a textual representation. The textual representation for the symbol table in the Jacobi solver example, not including metadata, is shown in Program 2.

Program 2 Symbol table for Jacobi solver example.

```
var Array2D<...> b;
var Array2D<...> _f6;
var Scalar<...> D;
var Array2D<...> _f5;
var Array2D<...> _f8;
var IterationVar<...> i;
var Stencil<...> R;
var Array2D<...> x;
var Array2D<...> r;
var ScalarConstant<...> num_iter;
var Array2D<...> _f7;
```

Semantic Model

The semantic model contains operations and loop constructs that specify the behavior of the program. Computational operations take variable names as arguments. Each variable name must have a corresponding entry in the symbol table. Operations are connected in sequential chains. Loops specify an iteration variable, an upper bound, and a body that is executed iteratively. Program 3 shows the semantic model for the Jacobi solver example.

5.6 Specifying Operations

Operations are implemented with one or more *generalized OpenCL kernels*. These kernels are mapped to the iteration space shown in Figure 5.2. The two middle loops, *nblocks* and

Program 3 Semantic model for Jacobi solver example.

```
for i 10
{
  Stencil2DMVM<op="Mult" >( _f5 R x);
  Array2DOp<op="Sub" >( _f6 b _f5 );
  Array2DCopy<>( r _f6 );
  ScalarArray2DOp<op="Mult" >( _f7 D r);
  Array2DCopy<>( x _f7 );
};
Array2DCopy<>( _f8 x);
Return<>( _f8 );
```

```
parfor d in 0..ndevices  // Added
  parfor b in 0..nblocks
    parfor i in 0..nworkitems
      for e in 0..nelements  // Added
```

Figure 5.2: Iteration space for our OpenCL kernels. We extend the traditional OpenCL launch with two extra loops: a parallel loop (d) over all devices and a serial loop (e) within a work item.

nworkitems, are built into an OpenCL kernel launch. Generalized kernels include the following additional loops. The *ndevices* loop to allows either for parallelization across multiple OpenCL devices or for sequential blocking for a device's last-level cache. Previous implementations of our framework supported parallelization over multiple devices, but support for multiple devices has been removed, and the *ndevices* loop is executed sequentially over one device instead. The inner loop, *nelements*, allows work items to process multiple data elements.

The programmer can specify one of several levels of the iteration space in which to specify an operation:

- **Device Level**: The operation is granted complete control over an OpenCL device. This allows the efficiency programmer to generate and enqueue multiple OpenCL kernels without interference from the framework.
- **Block Level**: The operation simply provides an OpenCL kernel to execute.
- **Item Level**: The operation is inserted into the second-inner-most loop (nworkitems). The operation may use OpenCL's built-in constructs to communicate among work items, such as barriers and local memory.

- **Element Level**: The operation is inserted into the inner-most loop (nelements). There may be no communication across parallel elements at this level.

Each operation invoked in the semantic model may read or write any argument that is passed to the operation, with the exception that operations may not write to variables of type *ScalarConstant* or *IterationVar*.

One or more generalized kernels are generated at runtime for each operation. Code generators have access to runtime knowledge such as datatypes, array dimensions, or stencil structure. Each generalized kernel must provide the following information:

- **Level**: The level at which this generalized kernel is specified (ie Device, Block, Item, or Element).
- **Source List and Sink List**: List of operand sources that are read from and sinks that are written to. If a variable is a sink that means it must be completely overwritten by the operation. Variables that are only partially overwritten should be added to both the source list and sink list.
- **Launch Parameters**: Details of the iteration space. Depending on the operation's level, launch parameters can include the number of devices (or device blocks), number of work items, number of work items per work group, and number of loop iterations per work item.
- **Emit function**: If the operation is specified at either the Element or Item level, it must provide a function called Emit(), which appends the OpenCL code for this operation to a C++ stringstream.
- **Execute function**: If the operation is specified at the Device level, it must provide a function called Execute(), which is called during execution. This function provides the generalized kernel latitude to execute multiple OpenCL kernels without interference from the framework.

Example

We demonstrate the specification of a 30×30 *Array2DOp* operation, which is a simple data-parallel array multiplication, addition, subtraction, or division where the array is a 30×30 Array2D. The OpenCL code we are aiming to generate by this operation is shown in the Program 8.

In addition to providing a function that generates this code, the operation must specify launch parameters, sources, and sinks.

Program 4 Generated OpenCL Code.

```
// Code here
for(int element_id1 = block_id1 * 16 + local_id1 ;
        element_id1 < min(128, (block_id1+1) * 16) ;
        element_id1 += 8)
{
  for(int element_id0 = block_id0 * 16 + local_id0 ;
          element_id0 < min(128, (block_id0+1) * 16) ;
          element_id0 += 8)
  {
    z[element_id0 + element_id1*30] = x[element_id0 + element_id1*30]
                                    + y[element_id0 + element_id1*30];
  }
}
```

Level

We chose to specify the *Array2DOp* operation at the Element level, meaning our OpenCL code will be inserted into the innermost loop of the OpenCL iteration space, specified in Figure 5.2.

Source and Sink List

The *Array2DOp* operation takes three arguments: the destination Array2D, **z**, and the two source Array2Ds **x** and **y**.

```
Array2DOp< op=``Sub'' >( z x y);
```

We annotate this operation to indicate that **x** and **y** are sources, and **z** is a sink. Each operation contains a list of source and sink operands, so these operands are simply appended to the lists. If we wanted to do an in-place Array2D operation then we would add the Array2D operand to both the source and sink lists. The following code adds operands to the source and sink lists and specifies partitions:

```
operation->addSource(
        new Array2DOperand(
        a1, Array2DDevicePartition( devsize, blksize, lp2.ne, 1)));
operation->addSource(
        new Array2DOperand(
        a2, Array2DDevicePartition(devsize, blksize, lp2.ne, 1)));
operation->addSink(
```

```
new Array2DOperand(
a3, Array2DDevicePartition(devsize, blksize, lp2.ne, 1)));
```

Launch Parameters

Launch parameters specify how many devices or device blocks, how many work items to launch, how many work items per work group, and how many loop iterations per work item. It can be advantageous for operations to have matching launch parameters to enable fusion, which is described in Section 5.7. We provide the productivity programmer a way of suggesting launch annotations for this reason.

This operation is performed on a 30×30 input vector. The operation has access to these dimensions when it is determining the launch parameters. We choose to decompose the 30×30 *Array2DOp* operation into a 2D launch over one device, a 2×2 grid of work groups, each with an 8×8 grid of work items, each item iterating over two elements in each dimension. We also specify an upper bound to the number of elements in the iteration space, in this case 30. This specification is encoded in the launch parameters shown in Table 5.1.

Devices	Work Groups	Items per Group	Elements per Item	Max elements
(1,1)	(2, 2)	(8,8)	(2,2)	(30,30)

Table 5.1: Launch parameters for *Array2DOp* operation.

OpenCL is able to reject launch parameters if it does not have the resources to support a specific configuration. For example, each device may have a maximum number of work items for each dimension, or a maximum number of work items per work group, that are reported by special OpenCL query functions independent of a specific kernel. OpenCL is also allowed reject configurations for specific kernels at runtime for any number of reasons. The framework currently does not handle the case where specific kernel configurations are rejected and must be reformed. The Intel, NVIDIA, and AMD platforms that we used for this work did not reject specific kernels in any of our applications, as long as we stayed within the bounds defined by the OpenCL query functions. It is future work to extend the framework to handle rejected launch configurations.

Emit Function

The emit function takes a string stream argument and produces a code snippet to be inserted into the innermost loop of the OpenCL iteration space. In order to allow for the use of local memory, operands are not referenced directly. Instead, we use accessor functions such as *get()* that are specific to each datatype. In this case, since the operands are Array2Ds, the *get()* function takes two arguments, an index for each of the two dimensions of the Array2D being accessed. Since the operation is completely data-parallel, we simply pass in the built-in indices element_id0 and element_id1.

Program 5 Sample emit function for the Array2DOp operation

```
void Array2DOpElementLevel::compile(std::stringstream & ss)
{
  Array2DOperand * dst = dynamic_cast<Array2DOperand*>(sinks[0]);
  Array2DOperand * src1 = dynamic_cast<Array2DOperand*>(sources[0]);
  Array2DOperand * src2 = dynamic_cast<Array2DOperand*>(sources[1]);

  ss << dst->get(``element_id0'', ``element_id1'') << ``='' <<
        src1->get(``element_id0'', ``element_id1'');
  if(op == ``Add'') ss << ``+'';
  if(op == ``Sub'') ss << ``-'';
  if(op == ``Mult'') ss << ``*'';
  if(op == ``Div'') ss << ``/'';
  ss << src2->get(``element_id0'', ``element_id1'') << ``;'' << std::endl;
}
```

This code will be executed at code generation time. The snippet may be included as its own kernel, or fused with other operations as described in Section 5.7. The two outer loops defining element_id0 and element_id1 are generated automatically by the framework and populated based on the launch parameters provided by this operation.

Execute Function

In this example we've chosen to specify the *Array2DOp* operation at the Element level, so we do not need to provide an execute function. Execute functions are specified for operations that need to take control of the entire device, potentially enqueuing multiple OpenCL kernels. Functions specified at the Element level are inlined, so only an emit function is required.

5.7 Fusion

Kernel fusion is a technique that increases the performance of CUDA and OpenCL code by reducing the overhead of a kernel launch, and by reordering operations to take advantage of producer-consumer locality through registers or cache. Figure 5.3 shows an example of fusing two vector operations: a vector addition followed by a squaring operation. Executing the fused kernel is more efficient than executing the two unfused kernels. There is only one kernel to launch instead of two, which reduces any effect of kernel launch overhead by a factor of two. Also, the fused kernel reads only two vectors **b** and **c** and writes one vector **a**. Executing the first unfused kernel would require two reads **b** and **c** and one write **a**. Executing the second unfused kernel would require one read **a** and one write **a**. Therefore,

```
__kernel void vvadd(__global float * a,
                    __global float * b,
                    __global float * c)
{
  a[get_global_id(0)] = b[get_global_id(0)] + c[get_global_id(0)];
}

__kernel void square(__global float * a)
{
  a[get_global_id(0)] = a[get_global_id(0)] * a[get_global_id(0)];
}

__kernel void fused(__global float * a,
                    __global float * b,
                    __global float * c)
{
  float t;
  t = b[get_global_id(0)] + c[get_global_id(0)];
  a[get_global_id(0)] = t*t;
}
```

Figure 5.3: Kernel fusion of a vector addition operation and a vector squaring operation

this fusion optimization reduces the total number of reads by one and reduces the total number of writes by one.

Fusion optimizations have historically been a feature of optimizing compilers [50]. Array accesses in dense loop nests are analyzed to form dependence vectors. Two consecutive loops are allowed to fuse as long as there are no fusion-preventing dependences between them. Loop fusion is not always profitable, for example if the register file becomes too full, so the decision of whether or not to fuse is often made by heuristics or search [108]. Fusion is also used to optimize groups of level-1 and level-2 BLAS operations. The build-to-order BLAS compiler [109] composes BLAS1 and BLAS2 operations dynamically, and searches over many legal fusion orderings, in order to produce code that runs faster than if the BLAS1 and BLAS2 operations are unfused. This approach has also been applied to CUDA-enabled GPUs [110].

Fusion optimizations are a key component of more recent GPGPU code generation frameworks. The high relative kernel launch overhead of discrete GPUs means fusion is often profitable, especially for operations with very low arithmetic intensity such as image processing kernels or data parallel primitives. Sato et al. describe a framework, embedded in the C language, that optimizes programs expressed with data parallel skeletons such as *map* and *zip* with fusion transformations [111]. Copperhead [61] also aggressively fuses data parallel

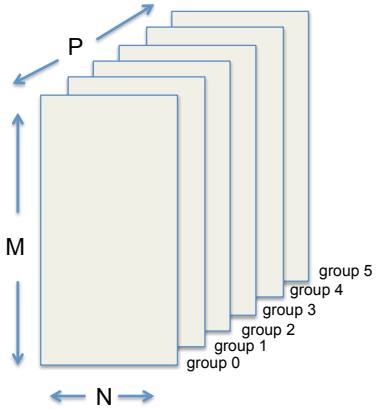


Figure 5.4: Example partitioning of a 3D array across work groups.

operations. In one example, for a sparse matrix-vector multiplication, Copperhead's fusion heuristic provides a 7.9× speedup over the unfused code.

Kernel fusion is not always possible. The legality of fusion depends on how each operation accesses its operands. In Copperhead, primitives are labeled with an "input completion" and "output completion" for different levels of the parallelism hierarchy, which can be either Local, Global, or None depending on the operation. A *map* operation, for example, labels both its inputs and outputs as Local, and is able to compose with other operations whose operands are labeled as Local. Certain sequences of operations can be fused, and others require synchronization between them.

Kernel Weaver [105] is a framework that exploits kernel fusion for relational algebra operations, such as *join* and *select*, which are common in database computations. Kernel Weaver's fusion approach is slightly different than Copperhead's. Kernel Weaver defines three types of dependence: thread dependence, cooperative thread array (CTA) dependence, and kernel dependence. Thread dependence is analogous to Copperhead's Local completion, meaning each thread owns one data element. CTA dependence implies that each thread block, in CUDA terminology, owns a collection of data elements from a partitioned input or output. Kernel dependence is analogous to Copperhead's Global completion, meaning that data is shared globally and a global synchronization may be required between operations. By defining three levels of dependence, they allow for fusion at multiple levels of hierarchy: either within a thread, within a thread block, or with global synchronization.

For the purposes of our framework, we felt it was important to allow for fusion at the level of the work group, or alternatively, the thread block. The reason for this comes from the case STAP study in Chapter 6. In the STAP application we have multiple operations on small matrices that we would like to fuse. Since we assign one matrix operation per work group, fusion would need to happen at that level. To enable this, we have the efficiency programmer partition all sources and sinks of each operation across work groups, if possible, and across work items, if possible. This is shown in Figure 5.4 for a 3D array. By specifying

this partition, the efficiency programmer is promising that each work group, or work item, will not modify data outside of its respective partition. If two operations both access the same data structure, they are allowed to fuse if they have the same launch parameters, and they divide the data structure similarly across that iteration space. In our implementation of the framework we only allow for simple partitionings, such as assigning same-sized contiguous chunks of an array to each work group. This provided sufficient flexibility for the applications we targeted.

The justification for this fusion technique in OpenCL follows from the OpenCL specification [37]. According to the OpenCL specification, there is load/store consistency within a work-item. Therefore, if we can guarantee a work-item has exclusive access to certain memory, then the work item can read and modify that memory repeatedly without any synchronization. The example code in Figure 5.3 shows a case where each work item has exclusive access to a single memory element for both operations and fusion is allowed. OpenCL also provides consistency of global memory across work-items in a single work-group assuming a local work-group barrier. Therefore, if we can guarantee that a work-group has exclusive access to certain memory, then the work-items in the work group can read and modify that memory repeatedly as long as there is a local barrier between read-after-write, write-after-write, and write-after-read accesses. Execution of kernels on an OpenCL device guarantees that all writes from the first kernel are completed on that device before the second kernel begins. In a multi-device setting, if we can guarantee that a device has exclusive access to certain memory then it can continue to enqueue kernels without performing synchronizations or sharing data with other devices. This suggests that operations should specify if and how operand data can be partitioned between work-items, work-groups, and devices (or device blocks).

To determine fusibility, each operation begins with a list of source operands and a list of sink operands. The framework maintains this list at every level of the parallelism hierarchy. As operations are fused at a given level, the source and sink list at that level are expanded to include sources and sinks from the incoming operation. Each source and sink operand is annotated with a partitioning. We use this information to determine if fusion is legal at a given level of the parallelism hierarchy. In order for two operations to fuse at a given level ℓ, any operands that the two operations have in common must have identical partitions for levels 0 to ℓ.

5.8 Dataflow Analysis

We would like to know how long each variable needs to be stored in memory in order to make memory optimizations. If certain data is only needed for one or two operations, we can use local memory or registers to store the data and avoid costly reads and writes from global memory. Without analysis, there is no way to know which variables are needed beyond a certain point in the program and which can be discarded. We use dataflow analysis to determine the live ranges of each variable.

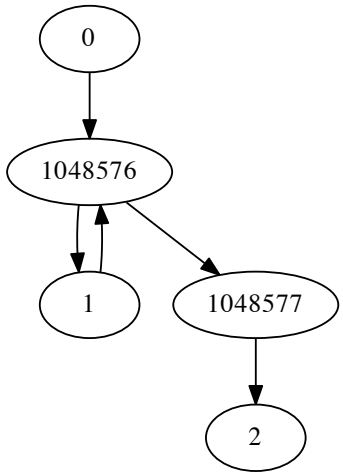


Figure 5.5: Basic blocks for the Jacobi solver example. They have been connected to demonstrate the possible flow of data between blocks. In this case, block 0 is empty, block 1048576 is the loop header, block 1 is the body of the loop, block 1048577 is the loop exit, and block 2 contains the return statement.

First we identify blocks of operations that contain no control flow, called basic blocks. Our intermediate representation only allows for two types of control flow: for loops and return statements. We walk through the intermediate representation and when a for loop is encountered, we end the current basic block and begin a new one inside the body of the for loop. Similarly, when a return statement is encountered, we end the basic block and begin a new one with the return statement.

In order to do dataflow analysis of the program, we must connect the basic blocks in a directed graph indicating possible ways in which data can flow. There are simple rules for how to connect blocks. Consecutive blocks are connected in program order. For loops are connected to allow for dataflow from the body of the for loop to itself, or to an empty block at the exit of the loop. This is shown in Figure 5.5 for the Jacobi solver example. Block 1048576 is the header of the for loop, and block 1 is the body. Block 2 is the return statement that comes directly after the loop.

We use the classic iterative dataflow algorithm [112] to determine the live-ins and live-outs for each basic block using the directed basic-block graph in Figure 5.5, along with the source and sink annotations for each operation. Once the live-ins and live-outs are determined for a given block, we can determine the live-ins and live-outs for each operation within the block using a simplified version of the dataflow algorithm.

The results of the dataflow analysis for the Jacobi solver example are shown in Table 5.2. The variable r is never used except as intermediate data between the *Array2DCopy*

Operation	live-ins	live outs
Stencil2DMVM	b, D, R, x	b, D, _f5, R
Array2DOp	b, D, _f5, R	b, _f6, D, R
Array2DCopy	b, _f6, D, R	b, D, R, r
ScalarArray2DOp	b, D, R, r	b, D, R, _f7
Array2DCopy	b, D, R, _f7	b, D, R, x

Table 5.2: Result of dataflow analysis for the Jacobi solver example. Operations from the body of the loop are shown.

and *ScalarArray2DOp* operations. It is not a live out of the for loop so it doesn't need to be stored in global memory or shared across devices at the end of each loop iteration. However, because x is used across iterations of the loop, it must be written to global memory or shared across devices in a multi-device setting.

Using Registers

Dataflow analysis tells us the live range of each variable in the program. In the Jacobi solver example, the variables _f5, _f6, _f7, and r are live for only two operations, meaning they are temporary variables. If r were to be used later, then it would remain live after the last operation.

One optimization is to use fast local memory to store temporary variables. In the example above, if the last two operations (ScalarArray2DOp and Array2DCopy) are fused at the element level, then r can be stored in registers and never written to global memory. To generalize this principle, we use the following rule: if a set of operations are fused at a given level ℓ, and a variable is neither a live-in of the first operation or a live-out of the last operation, then the variable can be stored in (promoted to) local memory at level ℓ.

In practice, our system only promotes variables to local memory in two specific cases. If a variable, such as an array, is partitioned such that there is only one word for each work-item, then we allocate a register for that word and use the register as temporary storage for the variable. Similarly, if a variable is partitoned such that there is only one word for each element-level loop iteration, then we allocate a register within the loop for that word and use the register as temporary storage.

5.9 Embedding in Python

Our framework is embedded in Python using a strategy similar to that of Copperhead [12] and Three Fingered Jack [14]: the productivity programmer expresses a function in Python; the framework is invoked at runtime, using a Python decorator, to process the Python AST and generate code; and then the code is executed and cached for later use. All inputs to

the program (e.g. the Jacobi solver's matrix, vector, and scalar inputs) begin as Python objects and data structures, and the framework returns its outputs as Python objects and data structures. In Copperhead, the productivity programmer expresses computations using a fixed set of data-parallel primitives. In Three Fingered Jack, the productivity programmer writes dense loop nests. Both of these frameworks accelerate a subset of Python; any Copperhead or Three Fingered Jack program is automatically a valid Python program. In contrast, our framework is designed so that the operations can be arbitrary functions. As a consequence, the efficiency programmer must create a separate Python implementation for every operation that produces the same solution as the accelerated implementation, in order for the program to be executable from Python.

Python Types

First, we need a way of representing each of the framework's types using Python objects. We use the Numpy [39] and Scipy libraries [40] for this purpose. Numpy provides array objects that are stored as contiguous C arrays and Scipy objects in turn use Numpy arrays for many data structures. Other objects, such as stencils, are custom data structures implemented in Python. Table 5.3 shows, for each of our framework's types, the corresponding Python object used to store each type's data.

Framework Data type	Python Base Class
Array	numpy.ndarray
Array2D	numpy.ndarray
Array3D	numpy.ndarray
SpMatDIA	scipy.sparse.dia_matrix
SpMatCSR	scipy.sparse.csr_matrix
SpMatDIA2D	Custom
Stencil	Custom
Scalar	numpy.{float32, float64, int64, bool}

Table 5.3: Our datatypes and their corresponding Python objects.

Runtime Compilation

When the Python function is called, the inputs to the function are examined and added to the symbol table. All inputs must have a type that is supported by the framework, ie a type listed in Table 5.3. The framework parses the Python function using Python's built-in parser. This produces an abstract syntax tree (AST) corresponding to the function's internal representation within Python. The framework proceeds to walk through the AST and flatten expressions using a standard generation technique for producing three-address

code [113]: when a supported operation is encountered, the operation is emitted and a temporary variable is produced to store the output and added to the symbol table.

The framework determines when an operation is supported by checking against a set of templates created by the efficiency programmer. There is one template for each supported operation. The template indicates the specific subtree the operation should match, including the types of each source or sink. Once an operation is matched, it is added to the semantic model. The template must also indicate the type of variable, if any, that is produced by the operation and provide a function to calculate the shape, size, and datatype of the new variable. The new variables are added to the symbol table as the AST is being processed.

Python Templates

Suppose the efficiency programmer creates an operation and would like to make it available to the productivity programmer in Python. The following templates can be used to embed operations in Python:

BinOp

One way to embed an operation is to use Python's built-in operators (+, -, *, /) to perform a two-operand operation, such as $A * x$. If an operation is to be embedded using the BinOp template, then it must specify which types it expects as the left and right operands and which binary op (+, -, *, or /) it expects in order to make a match. The BinOp template example below is a *Stencil2DMVM* operation, which expects a Stencil as the left operand, an Array2D as the right operand, and $*$ as the binary op.

```
BinOpTemplate('Stencil2DMVM',
              Variable({'type' : 'Stencil'}), # Left operand expected
              Variable({'type' : 'Array2D'}), # Right operand expected
              ast.Mult(),                     # Operation expected
              Stencil2DMVMInferFn)
```

CompareOp

In a manner similar to the BinOp embedding, operations can be embedded in the CompareOp node of the Python AST. In this case, the template for each operation must specify the types of the right and left operands and the specific compare op (<, >, <=, >=, ==, etc), if applicable. In the CompareOp template example below, the *ScalarCompare* operation specifies that it is expecting a Scalar type for both the right and left operands. The compare op it expects is left as *None*, which means it will match all available compare ops.

```
CompareOpTemplate('ScalarCompare',
                  Variable({'type' : 'Scalar'}), # Right Scalar operand
```

```
        Variable({'type' : 'Scalar'}), # Left Scalar operand
        None,                          # Compare op expected
        ScalarCompareInferFn))
```

Call

A Call node in the Python AST corresponds to a function call with a list of arguments, e.g. $sum(x)$. Call nodes provide a flexible way to embed an operation into Python because they can take any number of arguments. An operation embedded in a Call node must specify the function name to match. The example template below, for the *ArraySum* operation, specifies 'sum' as the name. The operation must also specify a list of operands and their types. In the case of *ArraySum*, there is only one operand and it is an Array.

```
CallTemplate('sum',                              # Function name
             'ArraySum',                         # Operation name
             [Variable({'type' : 'Array'})], # List of argument types
             ArraySumInferFn))
```

Assign

The Python expression $a = b$ is not allowed by the framework unless a and b are scalars. This is because aliasing between variables complicates analysis. However, assigning the value of one variable to another is used frequently during code generation. For example, to assign the variable y in the expression $y = A * x$, typically a temporary variable is created from the BinOp $A * x$, and the framework needs a way to copy the data from that temporary variable to y. So each variable type in Table 5.3 must also provide an operation and a corresponding template for copying the contents of that variable to another variable of the same type. The Assign template for Array2D is shown below. Here, the operation is *Array2DCopy* and it must only specify the type of variable being copied, in this case Array2D.

```
AssignTemplate('Array2DCopy',
               Variable({'type' : 'Array2D'}), # Right-hand side
               Array2DCopyInferFn))
```

Aliasing Rules and Restrictions

Since most variables are Numpy arrays, Scipy matrices, or custom objects, they are represented by references. Assigning one reference to another (e.g. r = x) would create an aliasing situation, where both r and x are pointing to the same memory. In order for a Python program to be processed by our framework, there must be no aliasing. Each reference name in the program points to a unique object in memory.

A common case is when an assignment is made (e.g. r = b - R*x), the right-hand side of the equation must produce an object of equal type, shape, and size as the variable on the left hand side of the equation. In Python, this statement would generate a new object and assign the reference r to point to that object. However, in our framework, we generate a new object and explicitly copy the value of that object into the memory reserved for the reference r. If the left hand side of the equation has a reference that is previously undefined, then it is defined with the type, shape, and size of the object that the right hand side produced.

5.10 Putting It All Together

We begin with the Jacobi solver example program, written in Python. The parameters to the *jacobi* function are D which is the value on the diagonal of the matrix, R which is a stencil representing the matrix without its diagonal, b which is the right-hand-side, and x which is the initial guess of the solution, and *num_iter* which is the number of solver iterations to execute. In this example, we set the number of solver iterations with an input variable, but the framework also supports if and break statements to allow for breaking out of the loop in the case of early convergence.

Program 6 Jacobi solver example application written in Python.

```
def jacobi(D, R, b, x):
  for i in range(num_iter):
    r = b - R*x
    x = D * r
  return x
```

The framework begins to process the *jacobi* function at runtime. The first step is to examine all the parameters in the header and add them to the symbol table:

```
var Scalar<dtype="float32" source="Header" > D;
var Stencil<dtype="float32" ndiags="4" source="Header" > R;
var Array2D<col_mjr="False" dim0="128" dim1="128" dtype="float32" source="Header" > b;
var Array2D<col_mjr="False" dim0="128" dim1="128" dtype="float32" source="Header" > x;
var ScalarConstant<value="10" > num_iter;
```

Next, the framework processes the abstract-syntax-tree (AST), checks each node against available templates, and produces a semantic model and any intermediate symbols necessary. For example, the first AST node to match a template is $R * x$ which matches the *Stencil2DMVM* operation. This is added to the symbol table, and an intermediate *Array2D* called *_f5* is produced to hold the result. At the end of this matching procedure, the symbol table and semantic model are as follows:

The next step is to apply the fusion approach from Section 5.7. The only basic block that has any operations is the body of the *for* loop. Given the sources and sinks of the operations

Program 7 Symbol table for Jacobi solver example application.

```
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Header" > b;
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Inferred" > _f6;
var Scalar<dtype="float32" source="Header" > D;
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Inferred" > _f5;
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Header" > _f8;
var IterationVar<dtype="int64" source="Inferred" > i;
var Stencil<dtype="float32" ndiags="4" source="Header" > R;
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Header" > x;
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Inferred" > r;
var ScalarConstant<value="10" > num_iter;
var Array2D<col_mjr="False" dim0="128"
        dim1="128" dtype="float32" source="Inferred" > _f7;


for i 10
{
  Stencil2DMVM<op="Mult" >( _f5 R x);
  Array2DOp<op="Sub" >( _f6 b _f5);
  Array2DCopy<>( r _f6);
  ScalarArray2DOp<op="Mult" >( _f7 D r);
  Array2DCopy<>( x _f7);
};
Array2DCopy<>( _f8 x);
Return<>( _f8);
```

in the body of the loop and their partitionings, the framework is able to fuse operations into two kernels. The fusion is shown in Figure 5.6, with the black boxes indicating OpenCL kernel barriers.

Next, we need to determine which variables must be saved in global memory and which can be stored only temporarily in registers. We run dataflow analysis, described in Section 5.8, to determine the live-ins and live-outs of each statement. These are shown in Table 5.2. This analysis enables certain non-obvious optimizations, such as storing the variable r in registers, which is shown in the generated code at the end of this section.

Finally, the *emit* function of each operation is called and the code shown in Program 8 is generated. This code is to be executed as the body of the *for* loop. Only the first kernel is shown.

5.11 Summary

In this chapter, we introduced an OpenCL composition framework, called Hindemith, that uses the selective embedded just-in-time specialization (SEJITS) methodology, along with

Program 8 Code generated for inner loop of Jacobi solver example application.

```
__kernel __attribute__((reqd_work_group_size(8, 8, 1)))
void k( const int device_id0,
        const int device_id1,
        __global float * _f6,
        __global float * b,
        __global float * D,
        __global float * _f5,
        __global float * R,
        __global int * R_offx,
        __global int * R_offy,
        __global float * x,
        __global float * r,
        __global float * _f7)
{
int block_id0 = device_id0 * get_num_groups( 0) + get_group_id(0);
int thread_id0 =  device_id0* get_global_size( 0) + get_global_id(0);
int local_id0 = get_local_id(0);
int block_id1 = device_id1 * get_num_groups( 1) + get_group_id(1);
int thread_id1 =  device_id1* get_global_size( 1) + get_global_id(1);
int local_id1 = get_local_id(1);
{
  for(int element_id1 = block_id1 * 16 + local_id1 ;
          element_id1 < min(128, (block_id1+1) * 16) ;
          element_id1 += 8)
  {
    for(int element_id0 = block_id0 * 16 + local_id0 ;
            element_id0 < min(128, (block_id0+1) * 16) ;
            element_id0 += 8)
      {
        float b_elt;
        float _f5_elt;
        float _f6_elt;
        float D_elt;
        float r_elt;
        float _f7_elt;
        {
          float msum = 0.0f;
          msum  += ((float)0.1) * x[(clamp(element_id0 + 0,0,127)) +
                                    (clamp(element_id1 + -1,0,127)) * 128];
          msum  += ((float)0.3) * x[(clamp(element_id0 + -1,0,127)) +
                                    (clamp(element_id1 + 0,0,127)) * 128];
          msum  += ((float)0.3) * x[(clamp(element_id0 + 1,0,127)) +
                                    (clamp(element_id1 + 0,0,127)) * 128];
          msum  += ((float)0.1) * x[(clamp(element_id0 + 0,0,127)) +
                                    (clamp(element_id1 + 1,0,127)) * 128];
          _f5_elt = msum;
        }
        {
          _f6_elt = b[(element_id0) + (element_id1) * 128] - _f5_elt;
        }
        {
          r_elt = _f6_elt;
        }
        {
          _f7_elt = *D * r_elt;
        }
        _f7[element_id0 + element_id1 * 128] = _f7_elt;
      }
    }
  }
}
```

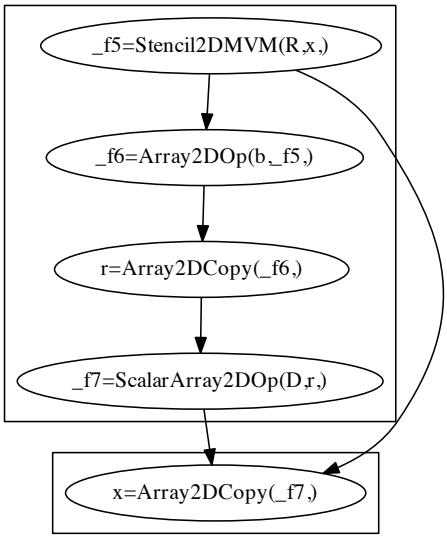


Figure 5.6: Demonstrating kernel fusion for the Jacobi linear solver example application. The rectangles indicate the boundary of an OpenCL kernel and each node in the graph is a different operation. In this case, the inner loop of the kernel is shown. It is fused into two OpenCL kernels.

high-performance hand-coded operations, organized around the principles of pattern-based software engineering. We identified four requirements for our framework:

- It is transparent and allows for custom hand-tuned code,
- It can take advantage of runtime knowledge such as shapes and sizes of data structures,
- It composes operations for efficiency, taking advantage of data reuse to reduce communication between processors and memory, and
- It is embeddable in a popular productivity language.

These four requirements are satisfied in the following manner. Hindemith is designed to be transparent, meaning that the efficiency programmer can provide operations with as much control over the hardware as OpenCL provides. Shape and size information are propagated through the program at code-generation time. The efficiency programmer also annotates the operations with dependence information, which is used by the framework to enable composition of operations for efficiency. Finally, the entire framework is embedded in Python.

Any program written in this framework has two components: the symbol table containing all data structures, and the semantic model, which contains operations on those data structures. Each operation in the semantic model must have a code generator, which is typically written by an efficiency programmer. This code generator emits generalized OpenCL kernels that are annotated with dependence and launch information.

The framework will compose, or fuse, generalized OpenCL kernels, do dataflow analysis to take advantage of registers, compile, and execute the code. The entire framework is embedded in Python, meaning that data structures begin and end as Python objects, and operations are invoked from specialized Python functions. In the next two chapters, we present case studies that use this framework for two applications: space-time adaptive processing, and optical flow.

5.12 Conclusion

The main challenge addressed by this chapter was how to design a framework that could allow for custom runtime-generated or hand-tuned code, such as the dense linear algebra operations described in Chapters 3 and 4, while providing a fusion mechanism that does more than could be accomplished with library calls. We allow operations to compose with one another, which requires us to annotate those operations with dependence information. The issue then becomes: will it be unprofitable to fuse nearby operations? For example, if two operations have been carefully tuned by hand and then they are fused, the result may perform worse. This is addressed by the two case studies in Chapters 6 and 7.

Chapter 6

Case Study: Space-Time Adaptive Processing

Chapters 6 and 7 are case studies using the Hindemith framework, described in Chapter 5, to solve two very different applications. The purpose of these case studies is to see if the framework is flexible enough to handle real applications, and also if the framework's fusion capabilities provide any performance gain. In this chapter we describe the implementation of a space-time adaptive radar processing application. This application was chosen for several reasons. First, it is an application that requires high performance for real-time use. Second, it has multiple levels of parallelism, both coarse and fine-grained, which makes achieving high performance a challenge.

6.1 Introduction

Space-time adaptive processing (STAP) is a technique for filtering out unwanted signals, such as ground clutter or jammers, from radar. This is accomplished by building an adaptive filter in real-time and applying the filter to the incoming radar data. Building an optimal filter is computationally infeasible for real-time applications, so suboptimal approaches are common [114]. In this chapter we implement and optimize the PERFECT Bench STAP benchmark, which is based on the suboptimal extended factor algorithm (EFA) [115].

We describe our implementation of the benchmark using the Hindemith framework introduced in Chapter 5. We evaluate our performance in terms of runtime, power, and energy compared to a hand-coded CUDA/CUBLAS implementation and a serial reference. For this particular application, our implementations exceed the performance of the serial reference by $10 - 23\times$, including all overheads such as PCIe transfer times. We also demonstrate how aggressive loop unrolling gives a large performance benefit for this application.

This chapter will focus on three computational kernels: the outer products kernel, system solve kernel, and inner-products kernel. For details of the underlying mathematics and signal processing techniques at work, refer to the PERFECT Bench documentation [116] or

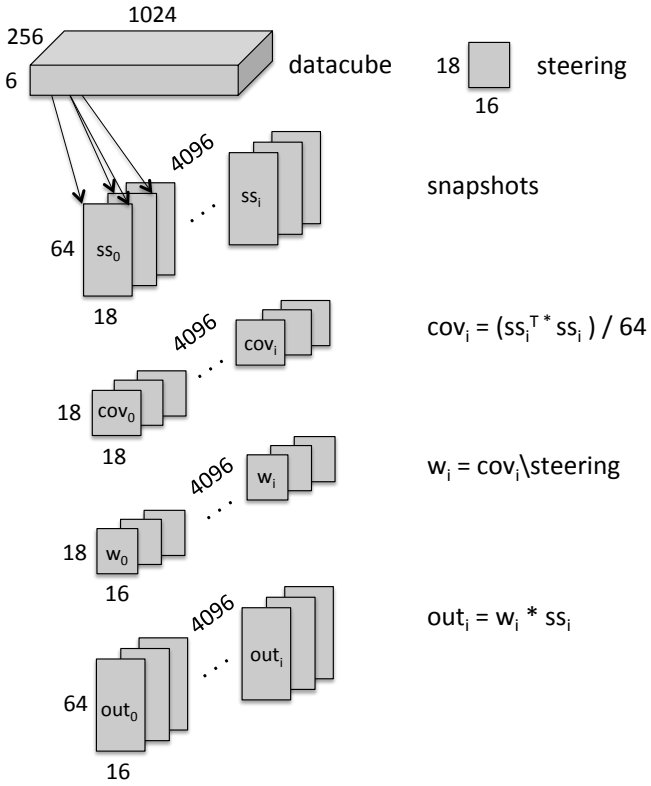


Figure 6.1: Key computational kernels for the STAP application

Klemm's textbook on space-time adaptive processing [117].

In related work, the STAP application has been evaluated on GPUs [118, 119] and accelerators [120]. The 1997 RT_STAP benchmark from the MITRE corporation is a real-time STAP benchmark that specifies three sets of problem parameters with varying computational complexity [97].

The following is a summary of the PERFECT Bench STAP application as described in the benchmark documentation [116]. The STAP computation is shown in Figure 6.1. Input data from the antenna array is stored in a 3-dimensional array called the datacube. The size of the datacube is the number of range cells (1024) by the number of antenna channels (6) by the number of pulses or Doppler bins (256). The range cell dimension is further divided into training blocks with 64 range cells in each block.

The first step is to extract snapshot vectors from the input datacube. There is a snapshot vector for every combination of range cell and Doppler bin. Each snapshot vector covers all six antenna channels and the two Doppler bins neighboring the Doppler bin of interest, for a total of 18 complex data values per snapshot vector. The snapshot vectors are coalesced into matrices by grouping snapshot vectors in the same training block. This results in 4096 snapshot matrices (number of training blocks times number of Doppler bins) of size 18x64.

Covariance matrices are computed for each snapshot matrix in the outer products kernel. These covariance matrices are passed to the system solve kernel. The system solve kernel generates unscaled weighting vectors by solving a 18x18 symmetric positive definite set of linear equations with with 16 right-hand-sides. The right-hand-side vectors are steering vectors, which correspond to specific radar tracking targets [98], and are the same for each of the 4096 independent problems. The resulting weight vectors are scaled by a constant and applied to the snapshot vectors in the inner products kernel. This produces a complex scalar output for every combination of range cell, Doppler bin, and steering vector. Like the snapshot vectors, the output data is coalesced into matrices by grouping range cells in the same training block. This results in 4096 output matrices (number of training block times number of Doppler bins) of size 16x64.

6.2 Implementation and Optimization

Figure 9 shows the STAP application written in Python for compilation by the Hindemith framework. All the data structures except the steering vectors are 3D arrays. Functions such as extract_snapshots and gamma_weights are specific for the STAP application and could not be reused. Other functions such as cgemm_batch and solve_batch are more general and could potentially be reused, however they currently only work for the shapes and sizes found in the STAP application. Scaling the covariance matrix may be reused in any other application where a 3D array is multiplied by a scalar.

Program 9 Hindemith code for STAP application

```
def stap(datacube, steering, scale):
  snapshots = extract_snapshots(datacube)
  covariance = cgemm_batch(snapshots)
  covariance = scale * covariance
  weights = solve_batch(covariance, steering)
  gamma = gamma_weights(weights, steering)
  output = inner_products(snapshots, weights, gamma)
  return output
```

Outer-Products Kernel

The outer-products kernel is implemented by the *cgemm_batch* operation in Program 9. This kernel requires a complex matrix multiplication (CGEMM), so we follow the approach used by both Volkov et al. [88] and the MAGMA BLAS library for GPUs [86]. For each of the 4096 small problems, we (the efficiency programmer) divide the 18x18 output matrix among the 64 parallel elements in a 2D cyclic manner. There are 8x8 work items per problem, so

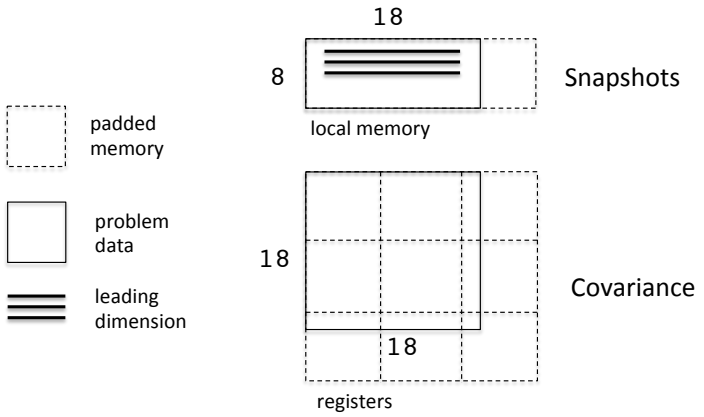


Figure 6.2: Data layout of outer products kernel. The snapshots matrix is stored in local memory in padded blocks of 24x8. Then the result of the matrix multiplication is accumulated into the covariance matrix, which is register allocated and padded. Each work item, of 64 total work items, has a 3x3 submatrix.

each work item owns a 3x3 register block of the covariance matrix. The snapshot matrix is loaded in blocks of size 8x18 and stored in a shared memory structure. The rows of the snapshot matrix are processed sequentially, and the data in each row is gathered by the work items so as to accumulate outer products of the snapshot vectors into the covariance matrix.

Since this operation is symmetric and uses complex arithmetic, the formula for the total number of FLOPs required is: $4 * M * N * K * np$, where M, N, and K, are the dimensions of the small matrices and np is the number of problems. In this case, $M = 18, N = 18, K = 64$, and $np = 4096$. So the total number of FLOPs is $4*18*18*64*4096 = 0.316$ GFLOPs. Our implementation ignores symmetry so it performs twice as many operations as is required.

System Solve Kernel

In this kernel we must solve 4096 small (18x18) symmetric positive definite linear systems with 16 right-hand-sides. The efficiency programmer specifies a custom operation that solves these systems. For each problem, the efficiency programmer specifices that the matrix should be loaded into registers distributed in a block-cyclic manner among work items, as described in Chapter 4. We use 64 work items in a 2-dimensional 8x8 layout, so each thread gets a 3x3 register block of the padded matrix, plus a 3x2 register block for the right hand sides. Together, this makes a 3x5 register block for the entire system plus right-hand-sides, as shown in Figure 6.3.

We load the covariance matrix into the first 3x3 register block, and the steering vectors into the 3x2 register block to the right of the matrix. Then we perform Gauss-Jordan elimination. For each k, from 0 to 18, we extract a row k from the matrix and right hand side data, and column k from the matrix. The extracted row is scaled by the inverse of the

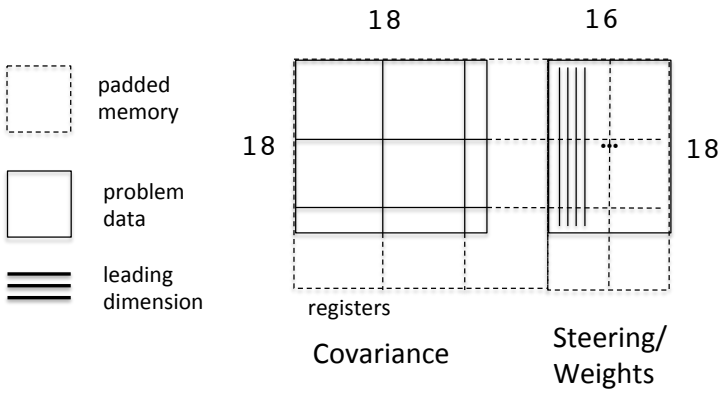


Figure 6.3: Data layout of system solve kernel

kth diagonal. This reduces the matrix to the identity. When the solve is complete, we store the right-hand-side vectors to the space in global memory reserved for the weight vectors.

This kernel can be solved using 4096 complex Cholesky factorizations (CPOTRF LAPACK routine) followed by triangular solves with multiple right hand sides (CTRSM). The formula for the total number of FLOPs is $((8/6) * N^3 + 8 * nrhs * N^2) * np$, where N is the dimension of the matrix, $nrhs$ is the number of right hand side vectors, and np is the number of problems. The N^3 term is for the Cholesky factorization, and the N^2 term is for the triangular solves. In this case, $N = 18, rhs = 16$, and $np = 4096$. So the total number of FLOPs is $((8/6) * 18^3 + 8 * 16 * 18^2) * 4096 = 0.187$ GFLOPs. Instead of using the Cholesky approach, we solve the system using Gauss-Jordan elimination, as described in Chapter 4. This requires more FLOPs, but it simplifies the implementation. The total number of FLOPs we are actually doing is $8 * N * N * (N + nrhs) * np = 8 * 18 * 18 * (18 + 16) * 4096 = 0.361$ GFLOPs.

Inner Products Kernel

We solve the inner products kernel in a similar manner to the outer products kernel, by distributing the output matrix across work items. However, in this case we load blocks of the input matrices of size 18x8 into the shared memory structure, as opposed to the 8x18 blocks we used for the outer products kernel. This layout is shown in Figure 6.4. We do this to improve spatial locality when loading the input matrices by loading in the leading dimension. We only store an 8x8 block of the output matrix at any given time, which is only one register per work item.

The inner products kernel requires 4096 complex matrix multiplications (CGEMM BLAS routine). We ignore the FLOPs required for gamma scaling because the operation is dominated by the matrix multiplication. The total number of FLOPs for the matrix multiplication is $8 * M * N * K * np$, where M, N, and K are the dimensions of the matrices and np is the

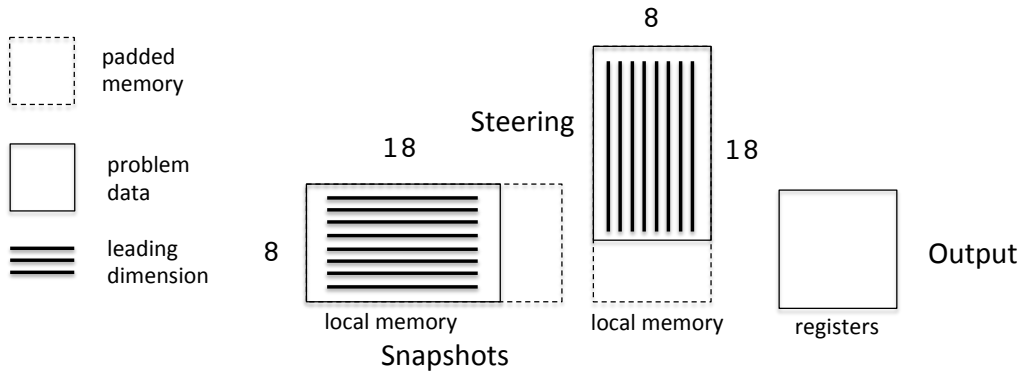


Figure 6.4: Data layout of inner products kernel

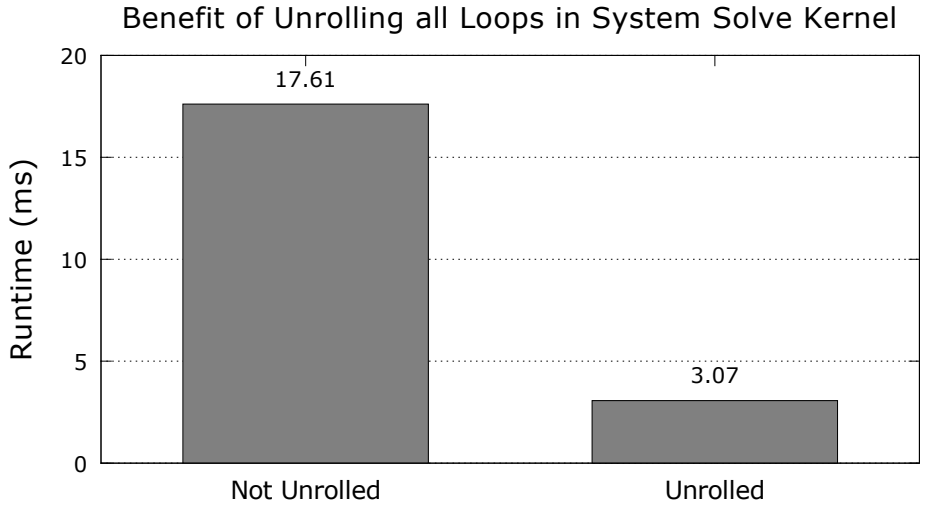


Figure 6.5: Runtimes for two versions of the System Solve kernel. The first version is written with loops executed at runtime, and the second (unrolled) version is written to unroll all loops at compile time and make all accesses to the matrix and right-hand sides statically known. This additional information makes it easier for the compiler to store the matrix in the register file.

number of problems. In this case, $M = 16, N = 64, K = 18$, and $np = 4096$. So the total number of FLOPs is $8 * 16 * 64 * 18 * 4096 = 0.562$ GFLOPs.

6.3 Performance Optimization by Unrolling

We observed a significant performance benefit for the STAP kernels, especially on NVIDIA GPUs, when we specified operations with unrolled loops. That is, the code-generator written by the efficiency programmer executes the loops during code-generation time and emits the body of the loop during each iteration. For example, we achieve a 5.7× speedup from unrolling loops in the system solve kernel on the NVIDIA GT 640 GPU, shown in Figure 6.5. The performance improvement has to do with the compiler's ability to register-allocate the matrix. If the loop is not unrolled then the matrix is stored in global memory, and hopefully cached in L1. If the loop is unrolled, then the matrix is stored in the register file.

The implementation, as specified by the efficiency programmer, divides the matrix across work items in a 2D cyclic manner for all three STAP kernels, as shown in Figures 6.2, 6.3, and 6.4. However, it is OpenCL's job to determine whether to store the matrix in registers or global memory. There are significant benefits to storing the matrix in the register file on NVIDIA GPUs. Since the register file is the largest on-chip cache, this allows for more matrices to be resident on the device at a given time. Also, data in registers is faster to access than in either cache or global memory.

The system solve kernel uses Gauss-Jordan elimination to solve for multiple right-hand sides of a small matrix. Gauss-Jordan elimination consists of three nested loops, described in detail in Section 4.3. In order for the compiler to register-allocate the matrix data, it must know which matrix elements are being accessed at compile time. In Gauss-Jordan elimination, the matrix elements are indexed using the induction variables of the three loops, which are not available until runtime. This means the compiler must generate loads and stores using these induction variables. We can completely expose the matrix indices at compile time by unrolling the two inner loops completely and unrolling the outer loop in blocks equal to the number of work items. Because the amount of code is $O(n^3)$ this approach is limited to small problems. This transformation and the ensuing performance gain is possible because we have knowledge of the matrix dimensions and we have the ability to programmatically generate code.

6.4 CUDA+CUBLAS Implementation

For comparison, we have a statically-compiled CUDA version of STAP that uses vendor-provided libraries. CUBLAS version 2, the vendor-provided BLAS library from NVIDIA, provides some batched routines that run efficiently on large numbers of small problems [121]. These routines include matrix multiply, LU factorization, and a matrix inversion that uses the result of the LU factorization. We use explicit matrix inversion instead of doing triangular solves (TRSM) because explicit inverse is significantly faster, despite requiring extra operations. CUBLAS does not provide a Cholesky factorization so we use LU instead. We disable pivoting for the LU factorization which improves performance. We use these routines, and three additional hand-coded CUDA kernels, to implement STAP. The pseudocode for

Program 10 CUBLAS routines and hand-coded CUDA kernels invoked for STAP

```
extract_snapshots <<< ... >>> (g_snapshots, g_input);
cublasCgemmBatched(..., 4096);    // Compute covariance matrices
cublasCgetrfBatched(..., 4096);   // Invert covariance matrices
cublasCgetriBatched(..., 4096);
cublasCgemmBatched(..., 4096);    // Compute weight matrices
gamma_weights <<< ... >>> (g_gamma, g_weights, g_steering);
cublasCgemmBatched(..., 4096);    // Compute output matrices
gamma_scale <<< ... >>> (g_output, g_tmp, g_gamma);
```

our CUDA STAP implementation is shown in Figure 10. The last parameter for the batched routines (4096) is the batch size, or the number of independent problems.

6.5 Performance

In this section, we report the performance of the STAP application on the following three platforms:

1. NVIDIA GT 640 GDDR5 mobile GPU with an AMD E-350 host processor. For these measurments, all computation takes place on the GPU device and the host is only used for running Python, compiling the OpenCL code, and enqueueing kernels. The GT 640 GDDR5 has 384 floating-point units running at 1.046 GHz and a pin bandwidth capacity of 40 GB/s. We are using CUDA version 6.0.

2. Intel Core i7-4770 Haswell quad-core processor running at 3.4 GHz with a pin bandwidth capacity of 25.6 GB/s. The 4770 also features AVX 2.0 vector instructions, an 8 MB last level cache. We are using the Beta version of the Intel SDK for OpenCL Applications 2014. Since this is a general purpose processor, the same device takes care of Python execution, kernel compilation, enqueueing kernels, and executing all operations.

3. AMD Radeon 7990 discrete GPU with a dual-socket Intel E5-2620 quad-core CPU host. All computation takes place on the GPU device and the host is only used for running Python, compiling the OpenCL code, and enqueueing kernels. The 7990 has two GPU ASICs on board but only one is used. The ASIC has 2048 floating-point units running at 950 MHz.

Runtime Comparisons

Runtime performance for the STAP application is shown in Figure 6.7 and Table 6.1. Compilation time is shown in Figure 6.6. The runtime is broken down into three categories:

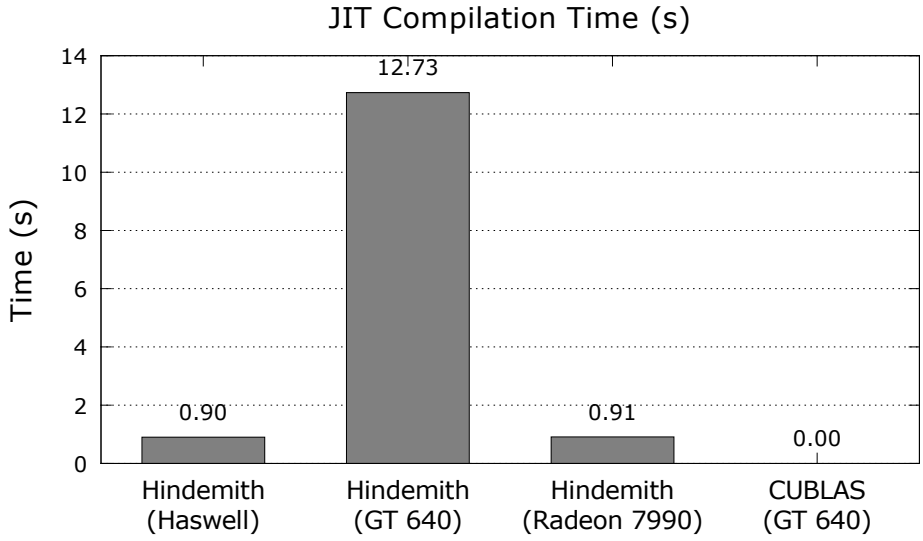


Figure 6.6: Time spent in JIT compilation phase for STAP on various machines

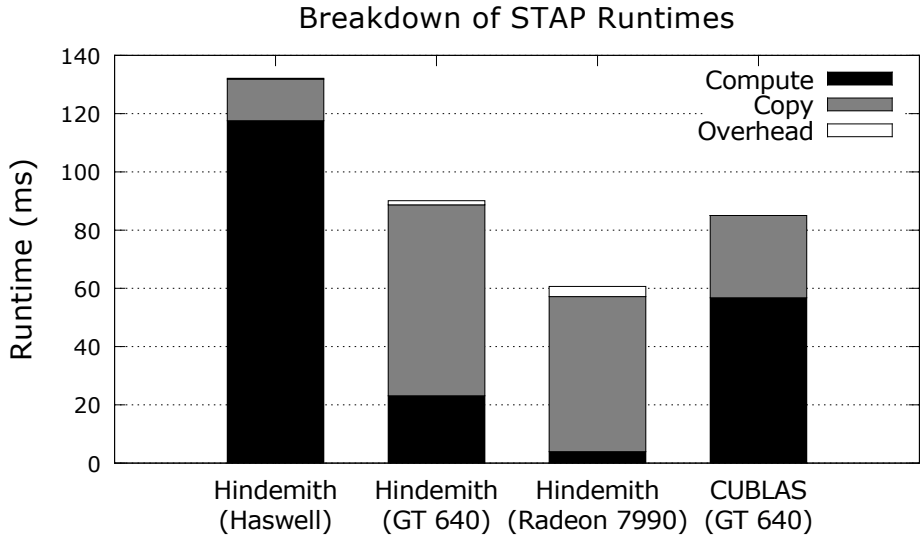


Figure 6.7: Runtime for STAP for Hindemith implementation on various machines, broken down into compute time, host/device copy time, and other overhead such as memory allocation and Python time. Also includes CUDA implementation using CUBLAS and a few hand-coded kernels. An unoptimized serial reference implementation that runs in 1.438 seconds on Haswell is not shown in this plot.

Implementation	Platform	Compute (ms)	Copy (ms)	Overhead (ms)	Total (ms)
PERFECT Reference	Haswell	1438	0	0	1438
Hindemith	Haswell	117.57	14.27	0.28	132.1
Hindemith	GT 640	23.11	65.56	1.43	90.12
Hindemith	Radeon 7990	3.93	53.25	3.48	60.66
CUDA+CUBLAS	GT 640	64.27	19.71	0	83.98

Table 6.1: Runtime for various implementations of STAP on different platforms

compute time, copy time, and overhead. The compute time is the time spent executing kernels on the device, after all data has been transferred to the device. Copy time is the time to transfer inputs to the device and outputs back to the host. Overhead is any additional time spent either in Python or the OpenCL runtime, for example on device memory allocation. Overhead is computed by subtracting the compute time and copy time from the total runtime as measured in Python.

The compute portion of the Hindemith STAP implementation outperforms the compute portion of the CUDA+CUBLAS version by a factor of 2.4× on the GT 640. This is not an ideal comparison, because the Hindemith STAP implementation is able to do the system solve kernel as one operation using Gauss-Jordan elimination, while the CUDA+CUBLAS implementation computes an LU factorization, then an explicit inverse, and finally uses matrix multiplication to complete the solve. The time spent transferring data over PCIe, the link between the CPU and GPU, is longer in Hindemith because we are not using pinned memory (ie cudaMallocHost). Pinned memory is used in the CUDA+CUBLAS version so the PCIe transfer time is reduced, which makes the performance of the CUDA+CUBLAS version roughly equal overall to the Hindemith implementation.

On the Radeon 7990, PCIe transfer times and other overheads far exceed the time spent on computation for STAP. The compute portion on the Radeon 7990 is only 4 milliseconds, while the time it takes to transfer the inputs and outputs over PCIe is over 50 milliseconds. In addition to the PCIe transfers, there are several milliseconds of overhead associated with allocating new device buffers to hold the output data. We observed variations in overhead time on the order of tens of milliseconds between successive executions of STAP.

The performance on Haswell is much lower than any of the GPU implementations. The code was written with GPUs in mind, and further tuned for GPUs as described in Section 6.3. We expect that this performance could be improved by tuning to ensure efficient use of Intel's SIMD hardware. The memory transfer time on Haswell was much lower than either of the GPU implementations because the data does not have to go over PCIe. In fact, we can completely eliminate this transfer by utilizing the CL_MEM_USE_HOST_PTR flag in clCreateBuffer. This is intended as future work. An unoptimized serial reference implementation also runs on the Haswell platform and has a runtime of 1.44 seconds.

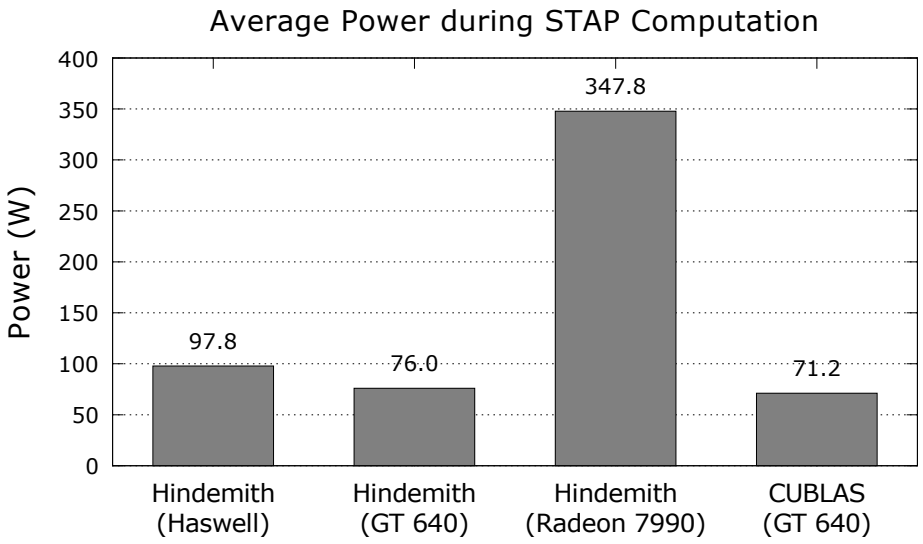


Figure 6.8: Average power consumption measured during the compute portion of STAP for Hindemith implementation on various machines. Also includes CUDA implementation using CUBLAS and a few hand-coded kernels.

Energy Comparisons

Figures 6.8 and 6.9 show the total energy and average power consumption for the compute portion of the STAP implementations on each platform. The average power consumption was measured using a "Watts Up? Pro" power meter [122]. The STAP compute portion was run in a loop for several seconds so that the power meter could take several samples which were then averaged. There were no PCIe transfers occurring in the compute loop. The total energy consumed per STAP execution was determined by multiplying the average power by the measured runtime excluding PCIe transfers.

The Haswell and the NVIDIA GT 640 platforms consume a similar amount of energy per second, between 71.2 and 97.8 Watts. The AMD platform was measured much higher at 347.8 Watts while computing STAP. Even though the AMD platform is several times faster than the GT 640 platform when executing the Hindemith implementation of STAP, the AMD platform is only 38% more energy efficient. Overall, the AMD platform running the Hindemith STAP implementation was the most energy efficient at 1.24 Joules.

Fusion

For this particular application, it is possible to use the Hindemith's fusion capabilities to merge all three kernels (inner products, system solve, and outer products) into a single OpenCL kernel. To do this, we adjusted the launch parameters so they are the same for all kernels. We executed the code first with both fusion and dataflow optimizations disabled,

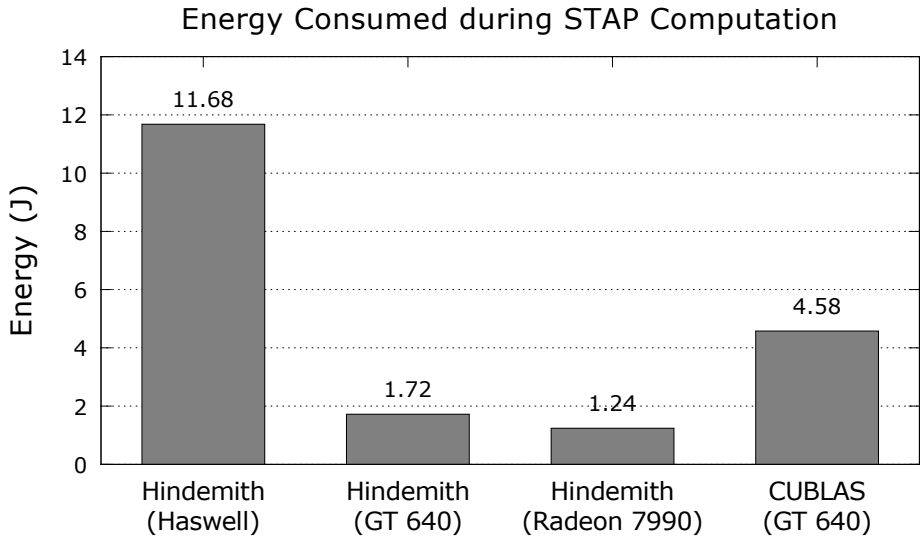


Figure 6.9: Energy of the compute portion of STAP for Hindemith implementation on various machines. Also includes CUDA implementation using CUBLAS and a few hand-coded kernels.

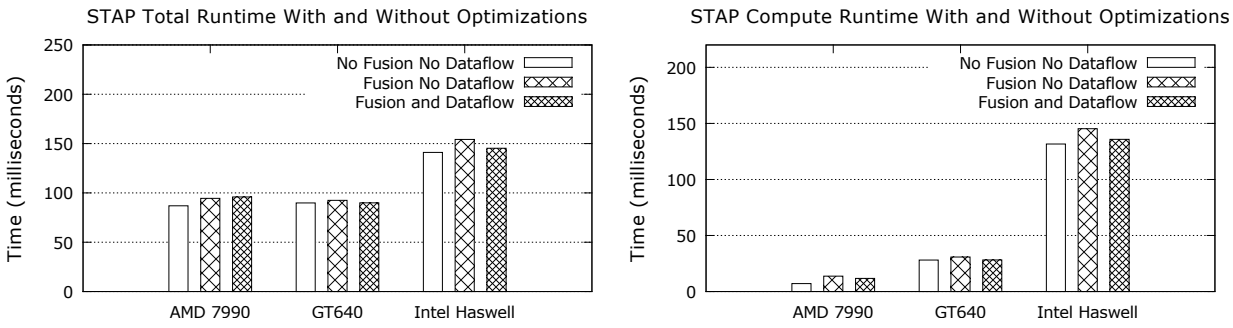


Figure 6.10: Runtime of STAP application with and without fusion and dataflow optimizations. We adjusted the launch parameters so that most operations could be fused into one large OpenCL kernel

then with only fusion enabled, then with both fusion and dataflow optimizations enabled. The runtime of the compute portion and the runtime of the total application are shown in Figure 6.10. The measurements on the AMD platform were noisy so we average over 10 consecutive executions. The optimizations did not provide a significant performance benefit on any of the platforms, and they sometimes hurt performance. For example, on the AMD GPU platform the runtime of the compute portion increased from 3.9 milliseconds in our original implementation to 11.7 milliseconds after adjusting the launch parameters to facilitate fusion. We could reduce the runtime of the compute portion on the AMD GPU platform to 7.1 milliseconds by disabling the fusion and dataflow optimizations.

6.6 Summary

In this chapter we implemented a space-time adaptive processing (STAP) benchmark using our OpenCL composition framework. The benchmark mainly consists of solving thousands of independent small matrix factorizations and matrix-multiplications. We examined these types of computations very thoroughly in Chapter 4, so the implementation in this chapter is very similar.

Embedding these small matrix operations into our framework essentially involved hand-coding the kernels (e.g system solve) and wrapping them as a framework operation. The advantage that the framework provides is that we can do code generation at runtime to unroll all loops. In the matrix operations described in Chapter 4, we unrolled all the loops using template metaprogramming which is not available for OpenCL. In this chapter, the dimensions of the matrix operations were hard-coded. However, the framework would allow us to extend these operations to handle different sized matrices.

We compared the performance of our STAP implementation to a C language reference, and an optimized CUDA + CUBLAS implementation. We ran the benchmark on three platforms: an NVIDIA GT640 GPU, an Intel Core i7 CPU, and an AMD Radeon 7990 GPU. On the Core i7 CPU, our implementation ran over 10× faster than the C language reference implementation. On the GT 640 GPU, the overall runtime was 8% slower than the CUDA + CUBLAS version. However, when considering only the compute portion, our implementation was 2.4× faster than the CUDA + CUBLAS version. For both GPUs, the time to copy from host to device memory was non-negligible. In the case of the AMD Radeon 7990 GPU, nearly all of the runtime was spent copying data to and from the device.

Finally, we tried to fuse all three kernels together and found that this either had no effect or caused a slowdown, especially on the AMD Radeon GPU.

6.7 Conclusion

The purpose of this case study was to see if the framework is flexible enough to handle real applications, an also if the framework's fusion capabilities provide any performance gain. In

this chapter, we showed that the framework was indeed flexible enough to handle the STAP application. The implementation we embedded in the framework exceeded the performance, in the compute portion, of the vendor libraries. In Chapter 4, we showed that for problems like these, a good percentage of the runtime was spent loading and storing matrix data. So one would expect that fusion of two operations would result in a speedup. However, when we fused all operations together in this case study it provided no benefit and sometimes hurt performance. This is likely due to the fact that the GPU's resources are allocated in a certain way for each operation, and when we fuse those operations resource allocations change. Since the factorizations were designed and optimized independent of one another, performance suffers when resource allocations change. This motivates an automatic, or guided search over all possible fusion combinations to determine the best configuration.

Chapter 7

Case Study: Optical Flow

Chapters 6 and 7 are case studies using the Hindemith framework, described in Chapter 5, to solve two very different applications. The purpose of these case studies is to see if the framework is flexible enough to handle real applications, and also if the framework's fusion capabilities provide any performance gain. In this chapter we describe the implementation of optical flow, a computer vision application. This application was chosen for several reasons. First, it is an application that requires high performance (e.g. 30 frames per second) for real-time use. Certain methods of optical flow also involve solving a sparse linear system of equations where the operator has a stencil structure. Optical flow can therefore be seen as representative of other applications in sparse linear algebra that require similar solvers.

7.1 Introduction

Optical flow is a common computer vision application that computes the displacement of each pixel between pairs of images, or between frames in a video. Optical flow between two images is visualized in Figure 7.1. Colors indicate direction and intensity indicates magnitude of pixel motion. Quality of solution is measured using standard benchmarks such as the Middlebury optical flow benchmark dataset [123] and the KTTI vision benchmark suite [124]. The quality metrics for optical flow are average angular error (AAE) of the flow vectors compared to ground truth provided by the these benchmark datasets, as well as average endpoint error (EPE) of the flow vectors. These error metrics are described further in Section 7.5.

There are many different ways to solve optical flow. As of February 2014, the KTTI vision benchmark results webpage for optical flow reports results for 42 different optical flow methods. The Middlebury benchmark results webpage reports results for 95 different optical flow methods. We choose two methods to focus on: Horn-Schunck [125] and Lucas-Kanade [126]. We focus on Horn-Schunck due to its popularity, along with the general consensus that many newer methods are simply extensions of this original formulation [127]. We include Lucas-Kanade in our analysis because it considers only local image patches, so

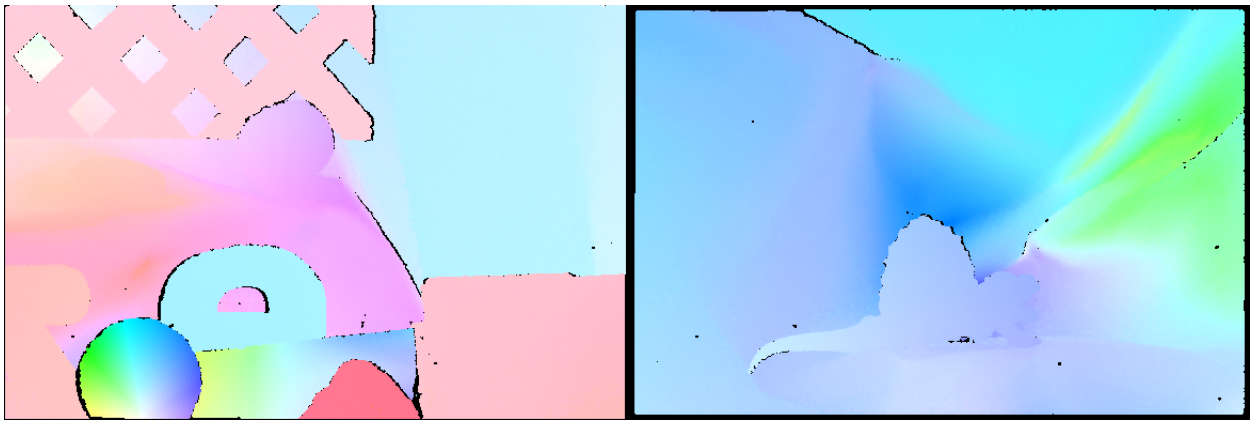
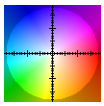


Figure 7.1: Visualization of ground truth optical flow from the Middlebury benchmark dataset, produced using the Middlebury visualization tools [123]. The RubberWhale benchmark is pictured on the left and the Dimetrodon benchmark is on pictured on the right.



(a) Colorcode mapping colors to flow directions and magnitudes. Figure reproduced from Baker et al [123].

it is fundamentally different than Horn-Schunck. The Lucas-Kanade method is also much cheaper to compute.

In Section 7.2, we give a more formal specification of the optical flow problem and describe two methods for computing optical flow in greater detail. Section 7.3 will describe the implementation of each method in our OpenCL composition framework. Section 7.4 will then show the computational performance (speed) of each method and compares the performance of different linear solvers on two different platforms. Section 7.5 lists the quality of solution in terms of AAE and EPE for our implementation of each method on two of the Middlebury benchmark images. In Section 7.6 we perform comparisons to other frameworks and implementations of optical flow. Then we measure the benefit of fusion optimizations in Section 7.7 and communication-avoiding algorithms in Section 7.8. Finally, Sections 7.9 and 7.10 summarize our results and present conclusions.

7.2 Problem Specification

For a pair of images I_0 and I_1, the optical flow problem is defined as finding functions u and v that minimize the following sum over all pixels [128]:

$$\sum_x \sum_y \left(I_1\left(x + u(x,y), y + v(x,y)\right) - I_0\left(x,y\right)\right)^2. \tag{7.1}$$

The functions $u(x,y)$ and $v(x,y)$ represent pixel displacements in the horizontal and vertical directions respectively, at the (x,y) location in the image.

Many optical flow methods, such as Horn-Schunck and Lucas-Kanade, begin modeling the optical flow problem by making a simplifying assumptions that lead to the following *optical flow constraint* [128]:

$$I_x u + I_y v = -I_t \tag{7.2}$$

In this linear equation, I_x, I_y and I_t are the derivatives of the image in the x, y, and time dimensions respectively. The unknown flow field is represented using vectors u and v, which are the same size as the image. In our implementations, we discretize the image derivatives I_x and I_y using the following filter:

$$\left(\frac{1}{12}, \frac{-8}{12}, 0, \frac{8}{12}, \frac{-1}{12}\right) \tag{7.3}$$

The optical flow equation is under-determined. There is one equation for each pixel and two unknowns (u and v pixel displacements) for each pixel. This is known as the *aperture problem*. Additional constraints must be introduced in order to solve this.

In the next subsections we describe two approaches to optical flow: the Horn-Schunck method and the Lucas-Kanade method. The Horn-Schunck method deals with the aperture problem by penalizing flow fields that are not smooth. The Lucas-Kanade method deals with the aperture problem by assuming that nearby pixels move together and solving for their movement using a least squares estimate.

Horn-Schunck

The Horn-Schunck method introduces a smoothness constraint, penalizing changes in the flow vectors across the image by adding a cost to the magnitude of the derivatives of the flow field. The formulation is an energy functional that should be minimized. The parameter α is included to trade off smoothness of the flow field with the accuracy of the flow nearby a given pixel. The integral sums this quantity for every pixel in the image. Each term in this functional represents a cost, so we try to minimize the functional over all possible flow functions u and v.

$$E = \iint \left(I_x u + I_y v + I_t\right)^2 + \alpha^2 \left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right) dxdy \tag{7.4}$$

Since $u(x,y)$ and $v(x,y)$ are functions, E is a functional. The Euler-Lagrange equations are used to find fixed points of this functional.

After plugging the appropriate terms into the Euler-Lagrange equations, the solution to the resulting linear equations provide a solution to the Horn-Schunck optical flow formulation. These linear equations use the Laplacians $\nabla^2 u$ and $\nabla^2 v$.

$$I_x^2 u + I_x I_y v = \alpha^2(\nabla^2 u) - I_x I_t \tag{7.5}$$

$$I_y I_x u + I_y^2 v = \alpha^2(\nabla^2 v) - I_y I_t \tag{7.6}$$

In our implementation, we also discretize the image derivative I_t and the partial derivatives of the flow vectors u and v using a $[-1, 1]$ filter. Using this filter appeared to be the simplest approach, but more complex filters could potentially produce more accurate derivative approximations. The term $\nabla^2 u = (u_{xx} + u_{yy})$ becomes a familiar 5-point stencil as shown in the following algebra. The stencil itself is shown in Equation 7.10.

$$u_x = u(x+1, y) - u(x, y) \tag{7.7}$$

$$u_{xx} = u_x - u_{x-1} = (u(x+1, y) - u(x, y)) - (u(x, y) - u(x-1, y)) \tag{7.8}$$

$$= u(x-1, y) - 2u(x, y) + u(x+1, y) \tag{7.9}$$

$$u_{xx} + u_{yy} = u(x-1, y) + u(x, y-1) - 4u(x, y) + u(x+1, y) + u(x, y+1) \tag{7.10}$$

Horn and Schunck suggested a slightly different discretization of the Laplacian as a 3×3 stencil [125]:

$$\begin{pmatrix} \frac{1}{12} & \frac{2}{12} & \frac{1}{12} \\ \frac{2}{12} & -1 & \frac{2}{12} \\ \frac{1}{12} & \frac{2}{12} & \frac{1}{12} \end{pmatrix}$$

We use the 5-point stencil in the following Horn-Schunck sections, and we use the 3×3 stencil when comparing to the ArrayFire implementation in Section 7.6.

Lucas-Kanade

As was described in the description of the Horn-Schunck method, the optical flow constraint equation is under-determined (the *aperture problem*) because there are two unknowns per pixel, u and v, and only one equation per pixel.

The Lucas-Kanade method [126, 129, 130] solves this problem by examining local regions across a pair of images and computing a displacement vector for each local region. The displacement vector for each local region is computed using a least squares solution to the optical flow constraint equation solved simultaneously for each pixel in the region. In order to solve for the flow vector, an overdetermined linear system is constructed using all the pixels in the surrounding region. This makes the assumption that all pixels in the region are moving in the same direction (same u and v). The vectors u and v are found using the

best fit in a least-squares sense. This procedure is done independently for each pixel in the image. Then, the algorithm iterates solving this least squares problem for each pixel, then warping the image (see Section 7.5) using the current estimate of the flow field, and solving again until a local minimum is found.

7.3 Implementation

Horn-Schunck

The code in Program 11 computes optical flow in Python using the Horn-Schunck method with a placeholder where the linear solver should be. This code can be specialized by our OpenCL composition framework. We've annotated the code with the input types.

First, du and dv are set to zero. Then, we compute the gradient images I_x and I_y by applying stencil operators Gx and Gy to the first input image. The variable I_t is computed as the difference between the first input image and the second image, warped based on the current estimate of the flow. Warping is used for multi-level optical flow, and described further in Section 7.5. If the current estimate of the flow is entirely zeros, then the warping procedure simply returns a copy of the second image and It stores the pixel-wise difference between the first and second images.

We do not do any preprocessing on the input images, such as smoothing. Any pixels accessed outside of the boundaries of the image, such as during a stencil operation, take their value from the closest pixel in the image. Extending the borders of the image in this way was the suggestion in the original Horn-Schunck paper [125].

The linear system arising from the Horn-Schunck method is symmetric and positive definite [131]. We solve the system using sparse iterative solvers such as Jacobi, conjugate-gradient (CG), and red-black Gauss Seidel (RB). We have implemented those linear solvers in our OpenCL composition framework, and we present performance and accuracy results in the following sections.

Jacobi Solver

The following listing shows our implementation of the Jacobi iterative linear solver. The positive definite stencil matrix is split into the diagonal and off diagonal parts. The diagonal elements are inverted and stored in $Dinv0$ and $Dinv1$. The off-diagonal elements are stored in the stencil matrix R and the vector $IxIy$ represents the coupling between du and dv. The off-diagonal parts of the matrix are applied to the current flow vectors using the following expressions: $R * du + IxIy * dv$ and $IxIy * du + R * dv$. The flow estimates du and dv are updated in each iteration according to the Jacobi iteration update rule [132].

Program 11 Horn-Schunck code with place-holder for linear solver

```
def hs_oflow    (im1_data, im2_data,  # Input images
                 R,                   # Laplacian stencil
                 Gx, Gy,              # Gradient stencils
                 u, v,                # Flow vectors
                 zero, one, lam2      # Scalars
                 ):
  du = zero * u
  dv = zero * v
  Ix = Gx*im1_data
  Iy = Gy*im1_data
  It = im1_data -
       warp_img2d(im2_data, u, v)

  ###
  # Insert linear solver here
  ###

  return du, dv
```

Program 12 Jacobi solver

```
  Dinv0 = one / (lam4 + Ix2)
  Dinv1 = one / (lam4 + Iy2)
  for i in range(num_iter):
    du_resid = b0 - (R * du + IxIy * dv) # b-R*x(k)
    dv_resid = b1 - (IxIy * du + R * dv) # b-R*x(k)
    du = Dinv0 * du_resid
    dv = Dinv1 * dv_resid
```

Iterative 2x2 Blockwise Solver

The algorithm provided by Horn and Schunck is an iterative solver based on the Jacobi method known as a 2×2 blockwise solver [131]. The code for this solver is shown below, and is written to match the example provided by ArrayFire [133]. The stencil matrix R holds only the off-diagonal elements. The estimates for the flow, du and dv, are updated in each iteration. It is possible to pre-compute $Ix * Ix$ and $Iy * Iy$, but this can actually slow down the solver by introducing more data structures and increasing the size of the working set.

Program 13 Iterative 2x2 Blockwise Solver

```
for i in range(num_iter):
  ubar = R * du
  vbar = R * dv
  num = Ix * ubar + Iy * vbar + It
  den = Ix * Ix + Iy * Iy + lam2
  du = ubar - (Ix * num) / den
  dv = vbar - (Iy * num) / den
```

Conjugate Gradient Solver

The code in Program 14 solves the Horn-Schunck linear system using the conjugate gradient method. We are essentially solving $Ax = b$, but the matrix is a coupled five-point stencil that is applied to two image-sized vectors simultaneously. The matrix is structured this way because it is a five-point Laplacian stencil, shown in Equation 7.10, applied to each component of the flow field, and the stencils are coupled by an off-diagonal ($IxIy$), shown in Equation 7.6. Since the framework doesn't have a special type for coupled stencils, we simulate this by representing the flow as two vectors, du and dv, and applying stencils and data-parallel operations to the two vectors individually. This makes the code more complicated. For example, to apply the coupled five-point stencil matrix to the vectors $p0$ and $p1$, we use several operations ($R * p0 + Ix2 * p0 + IxIy * p1$ to generate the first vector and $R * p1 + Iy2 * p1 + IxIy * p0$ to generate the second vector).

The conjugate gradient algorithm must exit early if the solution is found before the maximum number of iterations is reached. This is accomplished with the *if* and *break* statements in the inner loop. Because we check for convergence in the CG solver but not the other solvers, this gives a disadvantage in performance to CG compared to the other solvers. The reason we check for convergence in CG is to avoid numerical problems when the residual gets too small. The other iterative solvers did not have this problem.

Red-Black Gauss-Seidel Solver

The Gauss Seidel method is another iterative method of solving systems of linear equations [132]. However, the Gauss Seidel method is inherently serial because the vector is updated in-place, and the update to vector element i depends on previous updates to elements $0...i-1$. The Red Black ordering is a way of parallelizing Gauss Seidel. In this approach, the odd elements of a grid are updated independently of the even elements. Since there are no dependences between odd elements, all these updates can happen in parallel and vice versa for even elements. Our implementation of Red-Black Gauss Seidel is shown in Program 15. In order to allow for only "even" elements or "odd" elements to be updated, we created special functions *red* and *black*. These functions act like filters. The *red* function will allow all even elements to pass through, while setting the odd elements to zero. The *black* function

Program 14 Conjugate Gradient Solver

```
b0 = Ix * It
b1 = Iy * It
b0 = zero -  b0
b1 = zero -  b1
r0 = b0 - (R*du + Ix2*du + IxIy * dv)
r1 = b1 - (R*dv + Iy2*dv + IxIy * du)
p0 = b0 - (R*du + Ix2*du + IxIy * dv)
p1 = b1 - (R*dv + Iy2*dv + IxIy * du)
rsold = sum2d(r0 * r0 + r1 * r1)
for i in range(num_iter):
  Ap0 = (R*p0 + Ix2 * p0 + IxIy * p1)
  Ap1 = (R*p1 + Iy2 * p1 + IxIy * p0)
  alpha = rsold / sum2d(p0 * Ap0 + p1 * Ap1)
  du = du + alpha * p0
  dv = dv + alpha * p1
  r0 = r0 - alpha * Ap0
  r1 = r1 - alpha * Ap1
  rsnew = sum2d(r0 * r0 + r1 * r1)
  if rsnew < eps:
    break
  beta = rsnew / rsold
  p0 = r0 + beta * p0
  p1 = r1 + beta * p1
  rsold = rsnew
```

will do the same, but allow all odd elements to pass through and set even elements to zero.

Lucas-Kanade

The previous subsection was about Horn-Schunck, which is one method of solving optical flow. This subsection is about Lucas-Kanade, which is another method. The code in Program 16 computes optical flow using the Lucas-Kanade method. This code also runs in our OpenCL composition framework.

First we compute the gradient images Ix and Iy by applying the stencil operators Gx and Gy to the second image. Then we warp the second image and both the gradient images using the current estimate of the flow. This warping procedure is used when doing multi-level optical flow and when running the Lucas-Kanade approach for multiple iterations. We calculate the difference between the first image and the warped second image and store this in $ErrorImg$, which represents the error in the flow. Then we attempt to solve for the remaining error using the *lk_least_squares* operation. This operation performs a least squares fit of the overdetermined linear system containing pixels from the local region, for each pixel

Program 15 Red-Black Gauss-Seidel Solver

```
b0 = Ix * It
b1 = Iy * It
b0 = zero -  b0
b1 = zero -  b1
Dinv0 = one / (lam4 + Ix2)
Dinv1 = one / (lam4 + Iy2)

for i in range(num_iter):
  evens0 = red( Dinv0 * (b0 - (R * du + IxIy * dv)) )
  evens1 = red( Dinv1 * (b1 - (IxIy * du + R * dv)) )
  du = evens0 + black(du)
  dv = evens1 + black(dv)
  odds0 = black( Dinv0 * (b0 - (R * du + IxIy * dv)) )
  odds1 = black( Dinv1 * (b1 - (IxIy * du + R * dv)) )
  du = red(du) + odds0
  dv = red(dv) + odds1
```

Program 16 Lucas-Kanade solver

```
def LKSolver(I1, I2, Gx, Gy, u, v, zero, one, num_iter):
  Ix = Gx * I2
  Iy = Gy * I2
  WarpedI2 = warp_img2d(I2, u, v)
  WarpedIx = warp_img2d(Ix, u, v)
  WarpedIy = warp_img2d(Iy, u, v)
  ErrorImg =  I1 - WarpedI2
  du, dv = lk_least_squares(ErrorImg, WarpedIx, WarpedIy)
  return du, dv
```

in the image. The width of the window is specified as a constant before the function is called. We use a 9×9 window for our purposes in this section.

7.4 Performance

In this section, we consider the performance of our Horn-Schunck optical flow implementation in terms of runtime and operations-per-second. In Section 7.5 we discuss performance in terms of quality of solution. We also only report raw performance results in this section. Comparisons to other libraries and frameworks are considered in Section 7.6.

We measure performance on the following two platforms:

1. NVIDIA GT 640 GDDR5 mobile GPU with an AMD E-350 host processor. For these

Device	Solver	JIT time (s)	Compute + copy time (s)	# Iter.	Time per iteration (ms)	FLOPS per pixel per iteration	GFLOP/s (1e9)
GT640	HS Jacobi	8.02	0.186	400	0.465	24	11.7
	HS 2×2	7.92	0.158	400	0.395	30	17.2
	HS CG	8.68	0.608	400	1.519	48	7.2
	HS RB	8.45	0.458	400	1.145	28	5.5
	LK	1.65	0.016	1	0.017	810	11.7
Haswell	HS Jacobi	0.508	0.278	400	0.694	24	7.8
	HS 2×2	0.468	0.246	400	0.616	30	11.0
	HS CG	0.897	0.912	400	2.280	48	4.8
	HS RB	1.336	0.996	400	2.490	28	2.5
	LK	0.237	0.014	1	14.342	810	12.8

Table 7.1: Performance measurements for Horn-Schunck and Lucas-Kanade optical flow

measurements, all computation takes place on the GPU device and the host is only used for running Python, compiling the OpenCL code, and enqueueing kernels. The GT 640 GDDR5 has 384 floating-point units running at 1.046 GHz and a pin bandwidth capacity of 40 GB/s. We are using CUDA version 6.0.

2. Intel Core i7-4770 Haswell quad-core processor running at 3.4 GHz with a pin bandwidth capacity of 25.6 GB/s. The 4770 also features AVX 2.0 vector instructions, an 8 MB last level cache. We are using the Beta version of the Intel SDK for OpenCL Applications 2014. Since this is a general purpose processor, the same device takes care of Python execution, kernel compilation, enqueueing kernels, and executing all operations.

The solvers run twice in each Python session, as explained below. Each execution's runtime is measured with Python's *time* module.

1. In the first execution, we measure the combined JIT compilation and execution time. This is shown in Column 3 (JIT time).

2. The second execution measures the time to execute the optical flow solver combined with the time it takes to copy inputs and outputs to and from the device. We use a second set of inputs and outputs for this execution to ensure that inputs and outputs are not cached, and to ensure the copy time be included in the runtime measurement.

Table 7.1 shows the performance in terms of runtime and GFLOP/s for our Horn-Schunck and Lucas-Kanade implementations. We run each solver for a fixed 400 iterations, and

we verified that the CG solver does not exit early. This allows us to more easily analyze performance and the number of operations performed.

It takes between 1.65 and 8.68 seconds on the GT 640 and between 0.237 and 1.336 seconds on Haswell for the solvers to compile and execute the first time they are run. Since optical flow typically is run many times over a number of frames, we can consider this a one-time compilation overhead. The GT640 device takes much longer to compile than Haswell. The GT640's host is an AMD mobile CPU, which in our experience is very slow for both static and dynamic compilation.

We calculate the total number of floating-point operations (FLOPs) by counting operations in the inner loop of the solvers. The Jacobi, 2×2, CG, and RB solvers have 24, 30, 48, and 28 FLOPs in their inner loops respectively. To calculate the total number of FLOPs computed for the solvers, we multiply the number of FLOPs in the inner loop by the number of pixels, and divide by the time per iteration. This allows us to compute the rate of GFLOP/s in the last column of Table 7.1.

The best performing Horn-Schunck solver, in terms of GFLOP/s, is the 2×2, which achieves 17.2 GFLOP/s on the GT640 and 11.0 GFLOPs/s on Haswell. This is a simple solver in that it has no reductions (e.g. sum operations), and it performs only two global synchronizations per iteration. It also has the smallest working set of all three solvers, which improves performance on the cache-based 4770 platform. All of these factors contribute to the relatively high GFLOPs/s number. The conjugate-gradient solver, on the other hand, has several reductions per iteration and a larger working set, factors that we expect decrease performance on both platforms. The fact that we are checking for early convergence also reduces the performance of the CG solver compared to the others. The red-black solver implementation is inefficient because, the way we have it structured, it performs operations on all points and discards either the even or odd points depending on which are needed. We structured it this way so that we could reuse operations that operation on the entire array, instead of writing operations that only operate on even or odd points.

For Lucas-Kanade, the number of FLOPs per pixel is computed in the following manner. We are solving a least squares problem on a 9×9 region, meaning there are 81 equations and two unknowns. We solve least squares using the Normal Equations, meaning we have to compute $A^T A$, which takes $3 \times 2 \times 81$ FLOPs and $A^T b$, which takes $2 \times 2 \times 81$ FLOPs. This is a total of 810 FLOPs.

Different linear solvers also converge at different rates. Figure 7.3 shows the residual norm of the Horn-Schunck linear system for each linear solver over the course of 400 iterations for two benchmark images. In each case, the red black Gauss Seidel solver performs slightly better than either the Jacobi or 2×2 solvers. The conjugate gradient solver converges slowly at first, but then overtakes the other solvers after roughly 150 iterations.

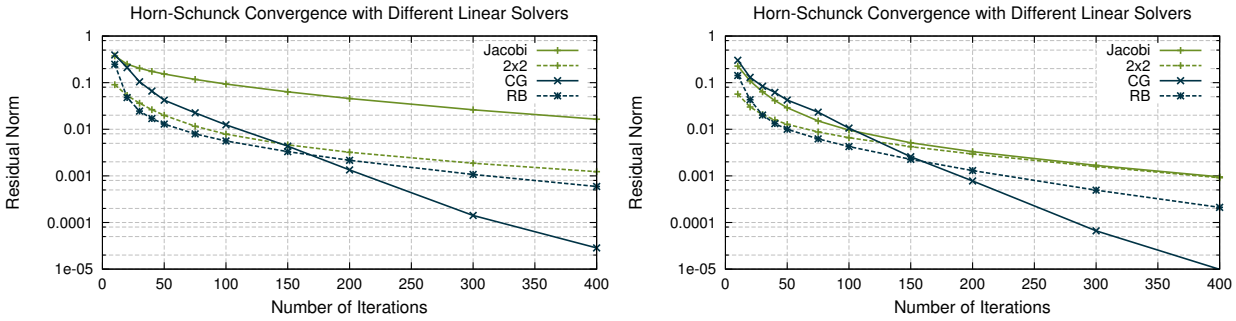


Figure 7.3: Convergence of different linear solvers for Horn-Schunck optical flow on the RubberWhale benchmark on the left and the Dimetrodon benchmark on the right

7.5 Quality of Solution

We consider the quality of solution of our Horn-Schunck optical flow implementation in terms of two quality metrics: average angular error (AAE) and end point error (EPE). The benchmark images we are running come from the Middlebury Optical Flow benchmark dataset [123]. Ground truth data is included along with the pairs of benchmark images. Given the ground truth data, we can compute the average angular error as the difference in the angle of the ground truth flow vectors and our computed flow vectors, averaged across all interior points. The average endpoint error is the distance between the end point of the ground truth flow vectors and our computed flow vectors, averaged across all interior points.

Multi-level

A common way to speed up and improve the quality of solution for many optical flow methods is to use a multi-level approach, also known as *hierarchical* or *coarse-to-fine* [128, 134]. This involves shrinking the image to a much smaller scale and solving optical flow on the smaller problem, then gradually resizing the problem larger and interpolating the flow solution at every level. We choose to resize images by a factor of two at every scale. The factor of two has been shown to produce roughly the same quality results as other popular downsampling factors [127].

Image Warping

At each level we perform a *warping* procedure. Bergen et al. describe image warping as "using the current values of the model parameters to compute a flow field, and then using this flow field to warp $I(t-1)$ towards I(t), which is used as the reference image ... The warped image (as against the original second image) is then used for the computation of the error ∇I for further estimation." [134]

Benchmark	Solver	Pyr.	Iter.	Median Filtering	JIT time (s)	Runtime (s)	AAE	EPE
RubberWhale	HS 2×2	1	400	Y	9.58	0.1758	12.77	0.59
RubberWhale	HS 2×2	1	400	N	6.48	0.5952	13.71	0.60
RubberWhale	**HS 2×2**	**4**	**400**	**Y**	**52.33**	**0.5077**	**11.07**	**0.26**
RubberWhale	HS 2×2	4	400	N	44.22	0.3078	12.77	0.31
RubberWhale	LK	1	-	Y	6.25	0.0355	13.52	0.56
RubberWhale	LK	1	-	N	3.18	0.5792	15.64	0.58
RubberWhale	LK	4	-	Y	44.38	0.0567	13.42	0.32
RubberWhale	LK	4	-	N	36.54	0.4698	18.13	0.47
Dimetrodon	HS 2×2	1	400	Y	11.36	0.1810	33.34	1.64
Dimetrodon	HS 2×2	1	400	N	6.54	0.1726	34.57	1.64
Dimetrodon	**HS 2×2**	**4**	**400**	**Y**	**57.67**	**0.5101**	**4.04**	**0.24**
Dimetrodon	HS 2×2	4	400	N	47.50	0.5049	5.09	0.30
Dimetrodon	LK	1	-	Y	4.81	0.0516	39.12	1.63
Dimetrodon	LK	1	-	N	3.22	0.0339	42.59	1.65
Dimetrodon	LK	4	-	Y	47.36	0.0596	5.35	0.27
Dimetrodon	LK	4	-	N	33.00	0.0517	7.28	0.37

Table 7.2: Performance and accuracy results for Horn-Schunck using a multi-level approach with and without median filtering. The most accurate configuration for each benchmark is shown in bold.

If the flow vectors are exactly correct, warping the first image should result in an exact copy of the second image. Typically the flow vectors are not exactly correct, so there is a residual error that must be corrected for. In a multi-level approach, we take the estimate of the flow derived from the smaller scales, use it to warp the image at the current larger scale, and improve the flow by solving for the residual error at the larger scale. We use bilinear interpolation in our warping implementation, which is the same interpolation approach used by Bergen et al. [134].

Median Filtering

Another useful technique is to employ a median filtering operation between each scale. This helps to smooth the flow field and eliminate outliers [127]. We use a 5×5 median filter and apply it to the flow field after resizing and interpolating the flow field to a larger scale. The median filter, resizing, and interpolation are all implemented as framework operations for performance reasons.

Results

Table 7.2 shows the performance and quality of solution results for optical flow using a multi-level approach described in Section 7.5. We compare the accuracy of both the Horn-Schunck and Lucas Kanade methods when we vary the number of levels in the multi-level approach, and whether or not the median filter is used. The best average angular error (AAE) and endpoint error (EPE) values are shown in bold for both benchmark images. In both cases it is the Horn-Schunck solver with 4 pyramid levels and median filtering enabled that performs

Figure 7.4: Visualization of flow field, produced using the Middlebury visualization tools [123], resulting from multi-level Horn-Schunck with median filtering. Two benchmark images are shown: RubberWhale and Dimetrodon. The color code on the right maps motion to colors. The colorcode figure is reproduced from Baker et al [123].

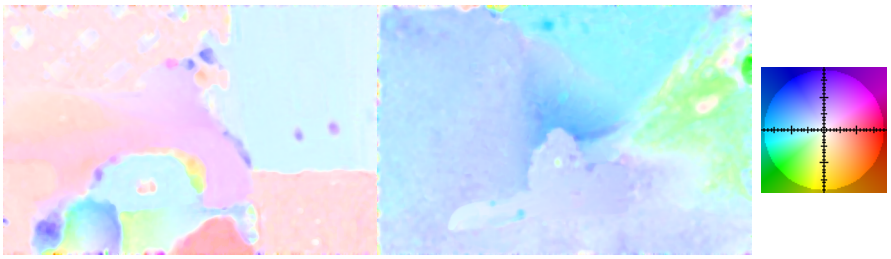


Figure 7.5: Visualization of flow field, produced using the Middlebury visualization tools [123], resulting from multi-level Lucas-Kanade with median filtering. Two benchmark images are shown: RubberWhale and Dimetrodon. The color code on the right maps motion to colors. The colorcode figure is reproduced from Baker et al [123].

best. The JIT time and runtime are also shown, running on the GT640 platform. The JIT time is much higher for the multi-level approach because the solvers are recompiled for each image size. The JIT time is especially high on the GT640 platform due to the low performance of the AMD E-350 host processor.

Figure 7.4 shows a visualization of the flow fields for the two benchmark images using the multi-level Horn-Schunck approach and Figure 7.5 shows the flow visualization for the Lucas-Kanade approach. The Horn-Schunck flow field looks smooth compared to Lucas-Kanade's flow field. This is because Horn-Schunck is a global method with a smoothness constraint built in. The Lucas-Kanade flow field is much less smooth due to the local nature of the method. Each 9×9 patch is solved independently, which can lead to small but sharp differences between nearby pixels.

7.6 Comparisons to Other Implementations

We contextualize the performance of our optical flow implementations by comparing our performance against other approaches. Our first comparison is against ArrayFire [55], a JIT-compiled array library that targets GPUs and multicore processors using OpenCL. Then, we compare to our own hand-coded implementations of Horn-Schunck optical flow, targeting a GPU using OpenCL.

Some comparisons required slight changes in the implementation to allow for an apples-to-apples comparison, such as how out-of-bounds pixels are handled and the structure of the stencils used. These differences arise from implementation details and are clearly noted in each section.

ArrayFire

ArrayFire is an array library for GPUs and multicore [55]. The library itself is embedded in C++ and provides special types, overloaded operators, and functions for arrays. ArrayFire also provides a JIT-compiler, which does lazy evaluation of array expressions and produces special OpenCL kernels at runtime. The developers of ArrayFire provide an implementation of Horn-Schunck optical flow as an example application [133]. This example application uses the iterative 2×2 block solver, described in Section 7.3. We extended the example to alternatively use Jacobi and CG solvers for comparison as well.

We modified our framework's implementation to match the implementation provided by ArrayFire in order to ensure a fair comparison with the ArrayFire library. For example, we use the following 3×3 stencil matrix instead of the 5-point stencil described in Section 7.2:

$$\begin{pmatrix} \frac{1}{12} & \frac{2}{12} & \frac{1}{12} \\ \frac{2}{12} & 0 & \frac{2}{12} \\ \frac{1}{12} & \frac{2}{12} & \frac{1}{12} \end{pmatrix}.$$

The ArrayFire convolution function (*convolve*) assigns zero values to all points outside of the image. Our framework's convolve function, in contrast, extends the value of the nearest border pixel as was the suggestion in the original Horn-Schunck paper [125]. However, for the purposes of comparison we wrote a special version of our stencil-vector multiplication that assigns zero values to all pixels outside of the border, as the ArrayFire implementation does. The zero-extending version is faster on the Haswell platform than the implementation that extended the border pixels. The stencils to compute the x and y gradients were set to match that provided by the ArrayFire example. Finally, we modified the ArrayFire calculation of the time gradient (It) array to be $It = I_2 - I_1$.

Figure 7.6 shows the runtime of our framework's implementation of Horn-Schunck for various linear solvers compared to ArrayFire on both the GT640 and Haswell platforms. We measure the ArrayFire runtime in the same way as our frameworks runtime: It runs once to

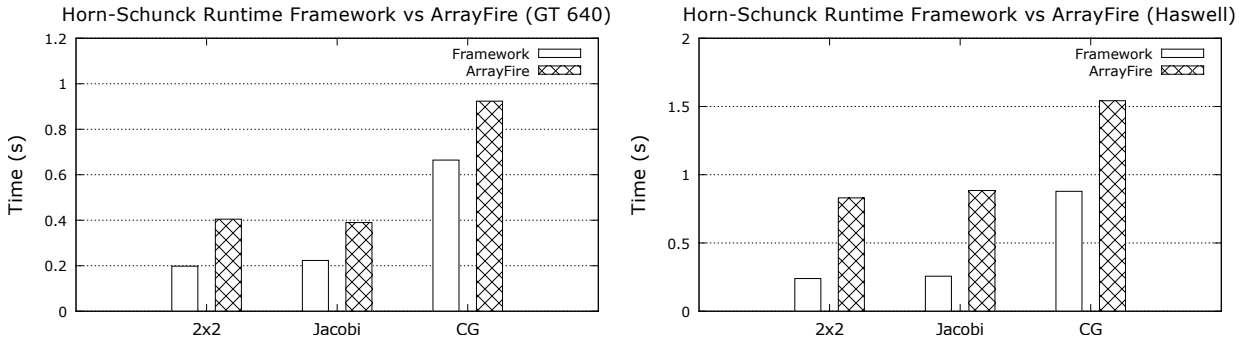


Figure 7.6: Runtime for Horn-Schunck, 400 iterations of various linear solvers, for our framework's implementation vs. ArrayFire implementation

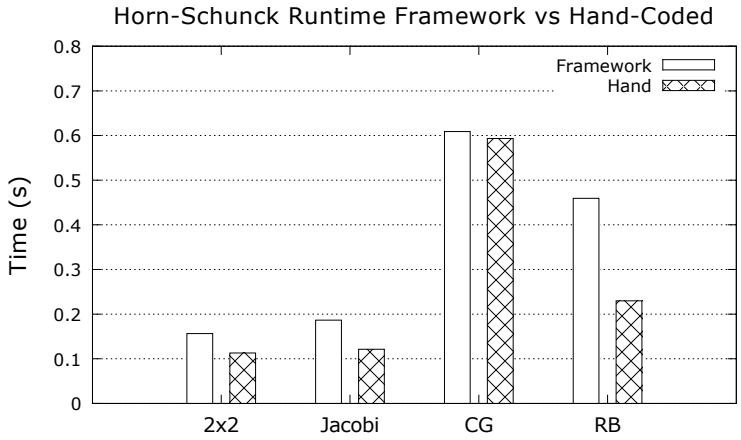


Figure 7.7: Runtime for Horn-Schunck, 400 iterations of multiple linear solvers for our framework's implementation vs. hand-coded implementation

allow for JIT compilation overhead and we measure the second run including host-to-device transfer times. The framework's implementation achieves between a 1.4× to 2× better frame rate than ArrayFire on the GT 640 platform and between 1.8× to 3.5× better frame rate on the Haswell platform.

Hand-coded

Next, we compare our framework's implementation of Horn-Schunck optical flow to a version that has been hand-coded in OpenCL. Both implementations extend border pixels rather than set them to zero, both check for early convergence in the case of CG, and both use the five-point stencil operator described in Section 7.2.

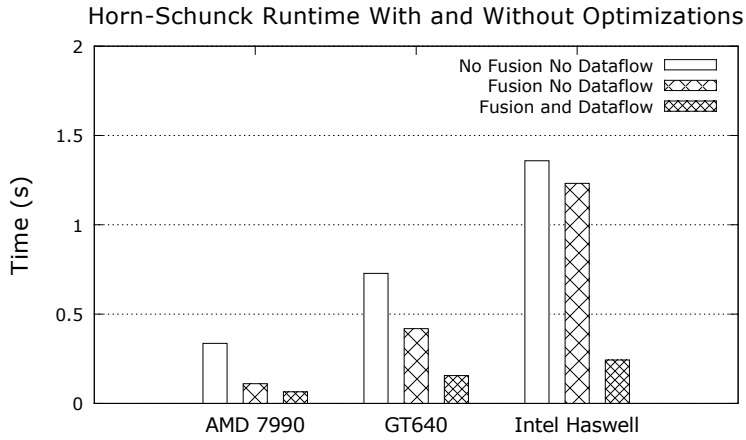


Figure 7.8: Runtime of Horn-Schunck 2×2 iterative solver with and without fusion and dataflow optimizations

Figure 7.7 shows the runtime of our framework's implementation of Horn-Schunck optical flow compared to the hand-coded implementation. This is for 400 iterations of various linear solvers on the GT 640 platform. We only count the time spent on computation so the input begins and ends on the GPU. The framework implementation achieves between $0.50\times$ and $0.97\times$ the frame rate of the hand-coded version. The hand-coded version uses a form of double-buffering for the Jacobi and 2x2 solvers, which cuts down on global memory traffic and synchronization compared to the framework's implementation. The hand-coded version of the red-black solver avoids doing redundant work on even and odd pixels, as opposed to the framework implementation, which does work on all pixels and then discards the half that are not needed.

7.7 Fusion

In this section we evaluate the effectiveness of the framework's fusion capabilities on Horn-Schunck optical flow using the 2×2 iterative solver. The details of these optimizations are described in Sections 5.7 and 5.8. In short, these optimizations improve performance by combining kernels to avoid kernel launch overhead, and storing data in registers rather than in global memory, which reduces memory traffic. We run the solver on three platforms: the AMD Radeon 7990, NVIDIA GT640, and Intel i7-4770. We run the solver for 400 iterations, use the 5-point stencil matrix, and extend border pixels rather than setting them to zero. These runtimes do include the time it takes to copy the data to and from the device.

We evaluate the following three settings: both fusion and dataflow optimizations disabled, fusion enabled and dataflow optimizations disabled, and both fusion and dataflow optimizations enabled. For this particular application, we see between a $4.6 - 5.5\times$ benefit to using these optimizations as opposed to disabling them in the framework. It should be

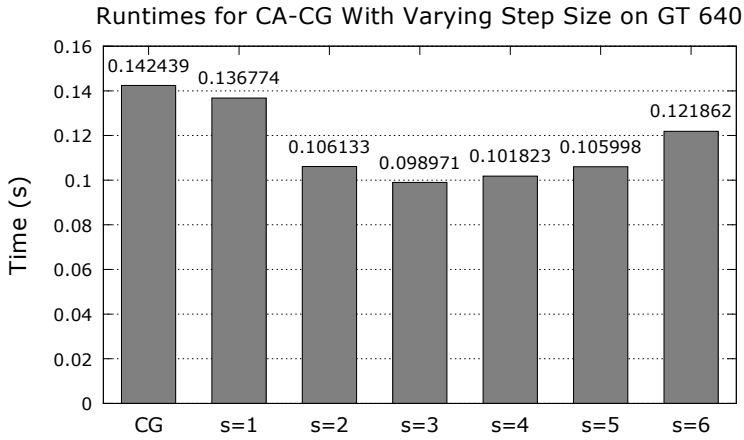


Figure 7.9: Runtime for Horn-Schunck optical flow using communication-avoiding CG with varying step sizes

noted that the framework introduces an extra array copy whenever an array assignment takes place, which is something that one wouldn't do if one were programming the application by hand. The fusion and dataflow optimizations then eliminate these unnecessary copy operations, and this contributes to these speedups.

7.8 Communication-Avoiding Algorithms

We consider the benefits of using communication-avoiding algorithms [32] for Horn-Schunck optical flow in this section. We created a hand-coded OpenCL implementation of communication-avoiding CG (CA-CG) algorithm using the monomial basis modeled after code from Carson et al. [135] and applied it to Horn-Schunck optical flow.

We compare the communication-avoiding implementation to another hand-coded implementation using the traditional CG algorithm. The traditional CG implementation is slightly different than that described in previous sections. We treat all boundaries of the image as zeros rather than extending border pixels. This makes for a simpler implementation of the matrix-powers kernel, a key part of CA-CG. Also, neither the traditional nor the communication-avoiding solvers check for early convergence. This alters performance slightly due to reduced synchronization and fewer CPU-to-GPU transfers.

Figure 7.9 shows the performance of traditional CG and CA-CG for 100 CG iterations, or equivalent, on the RubberWhale image on the GT 640 platform. The parameter s represents the number of steps taken during each inner iteration of the communication-avoiding algorithm. Increasing s enables more opportunity to avoid synchronization and data movement, but it also requires more work and more space in cache memories. This trade-off is shown in Figure 7.9. The best performance comes when $s = 3$, where the CA algorithm outperforms traditional CG by 43%. However, after 3, the costs of increasing s begin to outweigh the

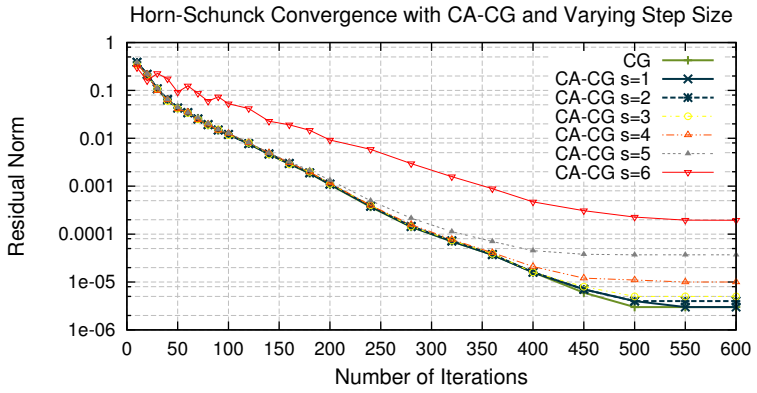


Figure 7.10: Convergence of communication-avoiding linear solvers for Horn-Schunck optical flow with varying step sizes

benefits.

Increasing s also has an effect on accuracy for this application. Figure 7.10 shows the residual norm for the CG and CA-CG solvers for Horn-Schunck optical flow and the RubberWhale image. There is very little accuracy penalty for increasing s from 1 to 4, and the result converges to roughly the same solution as the traditional CG algorithm. However, for $s = 5$ and $s = 6$, the solution begins to deteriorate by several orders of magnitude. These configurations do not provide any speedup anyway in our current implementation. However, we could potentially improve accuracy results by using 64-bit floating point arithmetic instead of 32-bit. Additionally, there are ways to combat this problem such as using a different basis for the Krylov subspace or modifying the CA-CG algorithm [135, 136].

7.9 Summary

In this chapter we implemented the optical flow application in our OpenCL composition framework. We implemented two optical flow methods: Horn-Schunck and Lucas-Kanade. For the Horn-Schunck method, we tried four different linear solvers: Jacobi, Iterative 2×2 blockwise, conjugate gradient, and red-black Gauss-Seidel. We measured performance on two different platforms: an NVIDIA GT640 and an Intel Core i7-4770. We also measured quality of solution for two benchmark images, using techniques such as multi-level, image warping, and median filtering.

We compared the performance of our framework to ArrayFire, a JIT-compiled array library for GPUs and multicore, and to our own hand-coded implementations. For the Horn-Schunck method with various linear solvers, the framework performed between 1.4× to 2× faster than ArrayFire on the GT 640 platform and between 1.8× to 3.5× faster on the Core i7 platform. The framework performed achieved between 0.50× and 0.97× the speed of our hand-coded version. We measured the effect of disabling fusion and dataflow optimizations. The optimizations provide between a 4.6× and 5.5× speedup compared to

the framework's performance of the same solver with optimizations disabled. Finally, we considered the benefit of using communication-avoiding algorithms for optical flow. We benchmarked Horn-Schunck optical flow using a communication-avoiding conjugate gradient solver. In its best configuration, the communication-avoiding implementation achieved a 43% speedup over traditional CG.

7.10 Conclusion

The purpose of this case study was to see if the framework is flexible enough to handle real applications, and also if the framework's fusion capabilities provide any performance gain. In terms of flexibility, the framework does very well in handling different approaches to optical flow, and allowing for different linear solvers to be used within the Horn-Schunck method. We implemented some basic sparse linear algebra operations, such as vector add and stencil-vector multiply, and this allowed us to implement several sparse solvers. The Lucas-Kanade implementation relied on operations that were specific to optical flow and generally not reusable. But it also used reusable components such as those to resize the image and perform median filtering.

The fusion and dataflow optimizations provided by the framework were particularly effective with this application. This is likely because all the operations were very light-weight and had low arithmetic intensity. The alternative to doing fusion and dataflow was to do a number of unnecessary reads and writes of vectors, which hurts performance. The operations were light-weight enough, and there were few enough of them, that the device resources were not overwhelmed and fusion did not cause a slowdown.

While we did get a speedup from using a communication-avoiding algorithm, the results were not as good as is likely possible. We expect that tuning the matrix powers kernel and adding more work per work group would be beneficial. Also, a cache-based platform would probably give better results than the GPU, which has an inverted memory hierarchy.

Chapter 8

Summary and Conclusion

Parallel processors have become ubiquitous and most programmers have access to parallel hardware, including multicore CPUs and GPUs. This has created an implementation gap, where efficiency programmers with knowledge of hardware details can attain high performance, while productivity programmers with application-level knowledge may not understand performance trade-offs.

In an effort to bridge the implementation gap, we propose a framework that is transparent meaning it allows for custom hand-tuned code, can take advantage of runtime knowledge such as shapes and sizes of data structures, composes operations to reduce communication between processors and memory, and is embedded in a productivity language. To this end, we deliver as a software artifact an implementation of Hindemith. The Hindemith framework uses the selective embedded just-in-time specialization (SEJITS) methodology, along with high-performance hand-coded operations, and was organized around the principles of pattern-based software engineering.

In Chapters 3 and 4, we demonstrated efficiency-level tuning for dense linear algebra applications. The particular shape and/or size of a problem can affect which algorithm or parallelization approach gives the best performance. Paying attention to hardware details such as the memory sizes and speeds provided improved performance on these applications. This experience also demonstrated the benefit of so-called "bare metal" tuning where the code was designed specifically to take advantage of hardware details such as memory sizes and speeds.

Guided by our experience optimizing these dense linear algebra operations, we identified four requirements for our framework:

- It is transparent and allows for custom hand-tuned code,
- It can take advantage of runtime knowledge such as shapes and sizes of data structures,
- It composes operations for efficiency, taking advantage of data reuse to reduce communication between processors and memory and

- It is embeddable in a popular productivity language.

These four requirements are satisfied in the following manner: Hindemith is designed to be transparent, meaning that the efficiency programmer can provide operations with as much control over the hardware as OpenCL provides. Runtime shape and size information are propagated through the program and rely on efficiency programmer annotations for each operations. The efficiency programmer also annotates the operations with dependence information, which is used by the framework to enable composition of operations to reduce communication between processors and memory. Finally, the entire framework is embedded in Python.

We evaluated the Hindemith framework using two application case studies: space-time adaptive processing (STAP) and optical flow. Ignoring JIT compilation time, our framework delivers optical flow frame rates that are between 0.5× and 0.97× our own hand-coded implementations. For STAP, the our implementation is competitive with a hand-coded CUDA + CUBLAS implementation.

The purpose of the case studies was to see if the framework is flexible enough to handle real applications, and also if the framework's fusion capabilities provide any performance gain. For both case studies, the framework was flexible enough to allow us to implement the applications. In the case of STAP, we essentially took a hand-optimized implementation of various matrix operations and wrapped them up as framework operations. For optical flow, we were able to implement very general sparse linear algebra operations, such as vector add and stencil-vector multiply, and reuse these operations for multiple optical flow implementations. This reuse is important because it takes longer to create an operation in the framework than it does to simply write an OpenCL kernel. It takes extra time to annotate all the operands. It also takes extra time to write a code generator instead of just the code itself, for example each operand can only be referenced through an accessor function. The idea is that this extra information and generality can improve performance by enabling fusion for example, and that the operation can be reused across applications.

The results of operation fusion were mixed. The framework's fusion optimization gave no performance benefit for the STAP application, and in some cases hurt performance. This was likely due to the fact that each operation was optimized separately, and combining operations changed the resource allocation to each operation. However, for the optical flow application, fusion provided a large speedup and was essential to achieving performance rivaling hand-coded versions. The optical flow case was different because the operations being fused were very light-weight and there were not so many of them that available resources were overwhelmed. The decision of whether or not to combine operations is not obvious, and is potentially a good opportunity for autotuning. This is analogous to the problem in optimizing compilers, where the profitability of loop fusion is not generally known.

8.1 Future Work

There are several opportunities for future work. First is the issue of our fusion heuristic. Currently our framework will attempt to fuse as many operations is possible without considering the profitability of such actions. We demonstrated in Chapter 6 that this can lead to dramatically reduced performance, and less fusion is sometimes better. This motivates future work into how to best make fusion decisions. For example, an automated or guided search over the space of all possible fusion combinations could potentially be effective.

Another opportunity for future work would be to expand the current list of supported operations. Currently, the framework supports a very basic set of vector, stencil, and sparse linear algebra operations needed for the optical flow solvers. It also has some custom operations written specifically for optical flow, such as image warping. There are some reusable image processing operations such as median filter. The operations used for space-time adaptive processing are less reusable. Future work would be to determine which operations are most useful for a wider range of applications and to provide implementations for these.

Bibliography

[1] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[2] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier/Morgan Kaufmann, Waltham, MA, USA, 5 edition, 2012.

[3] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[4] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, October 2009.

[7] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[8] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[9] Bryan Catanzaro, Shoaib Ashraf Kamil, Yunsup Lee, Krste Asanovi, James Demmel, Kurt Keutzer, John Shalf, Katherine A. Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. Technical

Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[10] Jike Chong. *Pattern-Oriented Application Frameworks for Domain Experts to Effectively Utilize Highly Parallel Manycore Microprocessors*. PhD thesis, University of California, Berkeley, 2010.

[11] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[12] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[13] Ekaterina I Gonina. *A Framework for Productive, Efficient and Portable Parallel Computing*. PhD thesis, University of California, Berkeley, 2013.

[14] David Sheffield, Michael J. Anderson, and Kurt Keutzer. Three Fingered Jack: Tackling portability, performance, and productivity with auto-parallelized Python. In *5th USENIX Workshop on Hot Topics in Parallelism, HotPar'13, San Jose, CA, USA, June 24-25, 2013*, 2013.

[15] Intel Corporation. Intel Core i5-4570 Processor (6M Cache, up to 3.60 GHz). `http://ark.intel.com/products/75043/Intel-Core-i5-4570-Processor-6M-Cache-up-to-3_60-GHz`.

[16] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[17] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[18] J. Nickolls and W.J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.

[19] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation Intel Core processor. *IEEE Micro*, 34(2):6–20, 2014.

[20] N. Kurd, M. Chowdhury, E. Burton, T.P. Thomas, C. Mozak, B. Boswell, M. Lal, A. Deval, J. Douglas, M. Elassal, A. Nalamalpu, T.M. Wilson, M. Merten, S. Chennupaty, W. Gomes, and R. Kumar. Haswell: A family of IA 22nm processors. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 112–113, Feb 2014.

[21] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[22] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[23] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, Aug 1996.

[24] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[25] Bret Toll, Elmoustapha Olud-Ahmed-Vall, and Ilya Albrekht. Intel Advanced Vector Extensions 2 and software optimization. *Intel Developer Forum*, 2013.

[26] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[27] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[28] John McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. *http://www.cs.virginia.edu/stream/*, 1995.

[29] David A Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.

[30] S. L. Graham, M. Snir, and C. A. Patterson, editors. *Getting up to Speed: The Future of Supercomputing*. Report of National Research Council of the National Academies Sciences. The National Academies Press, Washington, D.C., 2004.

[31] S. H. Fuller and L. I. Millett, editors. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, Washington, D.C., 2011. 200 pages, http://www.nap.edu.

[32] Mark Hoemmen. *Communication-Avoiding Krylov Subspace Methods*. PhD thesis, University of California, Berkeley, 2010.

[33] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[34] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[35] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[36] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, Mar 2009.

[37] Khronos OpenCL Working Group. The OpenCL specification Version: 1.0. 2009.

[38] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[39] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[40] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–.

[41] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[42] Mike Verdone. Python Twitter tools. `https://pypi.python.org/pypi/twitter`, 2014–. [Online; accessed 2014-09-12].

[43] The MathWorks Inc. MATLAB and Statistics Toolbox Release 2012b.

[44] Robert Fourer, David M. Gay, and Brian W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, May 1990.

[45] ILOG CPLEX 10.0 User's Manual, January 2006.

[46] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[47] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2), 2011.

[48] The MathWorks Inc. MathWorks Products and Services. `http://www.mathworks.co.uk/products/`.

[49] Brian W Kernighan and Dennis M Ritchie. *The C programming Language*. Prentice Hall, Upper Saddle River, NJ, 1978.

[50] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[51] NVIDIA Corporation. NVIDIA CUDA programming guide, 2009.

[52] Gary Bradski. The OpenCV library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.

[53] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[54] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[55] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU acceleration platform. In *SPIE Defense, Security, and Sensing*, volume 8403, pages 84030A – 84030A–8, 2012.

[56] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32, 2012.

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[58] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, volume 4, page 3, 2010.

[59] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for*

High Performance Computing, Networking, Storage and Analysis, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[60] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[61] Bryan Christopher Catanzaro. *Compilation Techniques for Embedded Data Parallel Languages*. PhD thesis, University of California, Berkeley, 2011.

[62] David Sheffield, Michael J. Anderson, and Kurt Keutzer. Automatic generation of application-specific accelerators for FPGAs from Python loop nests. In *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*, pages 567–570, 2012.

[63] H. Cook, E. Gonina, S. Kamil, G. Friedland, D. Patterson, and A. Fox. CUDA-level performance with Python-level productivity for Gaussian mixture model applications. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[64] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[65] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[66] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. Portable parallel performance from sequential, productive, embedded domain-specific languages. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 303–304, New York, NY, USA, 2012. ACM.

[67] NVIDIA Corporation. NVIDIA CUDA C programming guide version 6.0, 2014.

[68] Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.

[69] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, August 1977.

[70] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[71] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: Merging the PLPP and OPL projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[72] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[73] Aydın Buluç and John R Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, November 2011.

[74] Kaushik Datta. *Auto-tuning stencil codes for cache-based multicore platforms*. PhD thesis, University of California, Berkeley, 2009.

[75] Jeffrey Morlan, Shoaib Kamil, and Armando Fox. Auto-tuning the matrix powers kernel with SEJITS. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 391–403. Springer, 2013.

[76] E.J. Candes, X. Li, Y. Ma, and J. Wright. Robust principal component analysis. *Preprint*, 2009.

[77] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the 2009 ACM/IEEE conference on supercomputing*, SC '09, pages 36:1–36:12, New York, NY, USA, 2009. ACM.

[78] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May*, pages 2008–49, 2008.

[79] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 71–78. ACM, 2009.

[80] J.R. Humphrey, D.K. Price, K.E. Spagnoli, A.L. Paolini, and E.J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7705, 2010.

[81] James Demmel, Laura Grigori, Mark Frederick Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, EECS Department, University of California, Berkeley, Aug 2008. Current version available in the ArXiv at http://arxiv.org/pdf/0809.0101 Replaces EECS-2008-89 and EECS-2008-74.

[82] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Enhancing parallelism of tile QR factorization for multicore architectures. LAPACK Working Note 222, September 2009. UT-CS-09-645.

[83] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, September 2005.

[84] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langem. QR factorization of tall and skinny matrices in a grid computing environment. In *24th IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.

[85] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[86] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[87] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 121–130, New York, NY, USA, 2009. ACM.

[88] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[89] Grey Ballard, James Demmel, Laura Grigori, Mathias Jacquelin, Hong Diep Nguyen, and Edgar Solomonik. Reconstructing Householder vectors from tall-skinny QR. Technical Report UCB/EECS-2013-175, EECS Department, University of California, Berkeley, Oct 2013.

[90] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[91] University of Tennessee Innovative Computing Laboratory. MAGMA: Matrix algebra on GPU and multicore architectures. `http://http://icl.cs.utk.edu/magma/`. Accessed: 10/26/2014.

[92] NERSC. Experimental GPU cluster: Dirac. `http://www.nersc.gov/nusers/systems/dirac/`.

[93] R. Vezzani and R. Cucchiara. ViSOR: Video surveillance on-line repository for annotation retrieval. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 1281–1284, June 2008.

[94] X. Yuan and J. Yang. Sparse and low-rank matrix decomposition via alternating direction methods. *Preprint*, 2009.

[95] Y. Dong and GN DeSouza. Adaptive learning of multi-subspace for foreground detection under illumination changes. *Computer Vision and Image Understanding*, 115:31–49.

[96] M. Murphy, K. Keutzer, S. Vasanawala, and M. Lustig. Clinically feasible reconstruction time for L1-SPIRiT parallel imaging and compressed sensing MRI. In *ISMRM '10: Proceedings of the International Society for Magnetic Resonance in Medicine*, page 4854, 2010.

[97] K.C. Cain, R.T. Williams, and Torres J.A. RT_STAP: Real-time space-time adaptive processing benchmark. Technical report, MITRE Corporation, 1997.

[98] Michael Parker. Radar basics. `http://www.eetimes.com/design/programmable-logic/4216104/Radar-basics---Part-1`.

[99] P.R. Dixon, T. Oonishi, and S. Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Computer Speech & Language*, 23(4):510–526, 2009.

[100] Michael J Anderson, David Sheffield, and Kurt Keutzer. A predictive model for solving small linear algebra problems in GPU registers. In *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 2–13. IEEE, 2012.

[101] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[102] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved MAGMA GEMM for Fermi GPUs. LAPACK Working Note 227, July 2010. UT-CS-10-655.

[103] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, March 2010.

[104] Khronos group. OpenVX: Hardware acceleration API for computer vision applications and libraries. `http://www.khronos.org/openvx`, 2014–. [Online; accessed 2014-09-12].

[105] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel Weaver: Automatically fusing database primitives for efficient GPU computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE Computer Society, 2012.

[106] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[107] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[108] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 249–258, New York, NY, USA, 2006. ACM.

[109] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12, New York, NY, USA, 2009. ACM.

[110] J Filipovič, Matus Madzin, Jan Fousek, and Ludek Matyska. Optimizing CUDA code by kernel fusion-application on BLAS. *arXiv preprint arXiv:1305.1183*, 2013.

[111] Shigeyuki Sato and Hideya Iwasaki. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 79–94, Berlin, Heidelberg, 2009. Springer-Verlag.

[112] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[113] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[114] Wolfram Bürger. Space-time adaptive processing: Algorithms. *Advanced Radar Signal and Data Processing. Educational Notes RTO-EN-SET-086, Paper 7*, 2006.

[115] R.C. DiPietro. Extended factored space-time processing for airborne radar systems. In *Conference Record of the Twenty-Sixth Asilomar Conference on Signals, Systems and Computers*, pages 425–430 vol.1, Oct 1992.

[116] Pacific Northwest National Labratory and Global Technology Resources Inc. DARPA PERFECT Suite Documentation. Version 0.0.2, 2013.

[117] Richard Klemm. *Principles of Space-Time Adaptive Processing*. Number 159. IEE Press, 2002.

[118] Jimmy Pettersson and Ian Wainwright. Radar signal processing with graphics processors (GPUs). Master's thesis, Uppsala University, Division of Scientific Computing, 2010.

[119] T.M. Benson, R.K. Hersey, and E. Culpepper. GPU-based space-time adaptive processing (STAP) for radar. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[120] Sourav Chatterji, Manikandan Narayanan, Jason Duell, and Leonid Oliker. Performance evaluation of two emerging media processors: VIRAM and Imagine. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 229.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[121] NVIDIA Corporation. cuBLAS — NVIDIA developer zone. `https://developer.nvidia.com/cublas`.

[122] Watts Up? Watts Up? plug load meters. `https://www.wattsupmeters.com/secure/products.php?pn=0`. Accessed: 10/27/2014.

[123] Simon Baker, Daniel Scharstein, JP Lewis, Stefan Roth, Michael J Black, and Richard Szeliski. A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, 92(1):1–31, 2011.

[124] J. Fritsch, T. Kuhnl, and A. Geiger. A new performance measure and evaluation benchmark for road detection algorithms. In *Intelligent Transportation Systems - (ITSC), 2013 16th International IEEE Conference on*, pages 1693–1700, Oct 2013.

[125] Berthold KP Horn and Brian G Schunck. Determining optical flow. *Artificial intelligence*, 17(1):185–203, 1981.

[126] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[127] Deqing Sun, S. Roth, and M. J. Black. Secrets of optical flow estimation and their principles. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2432–2439. IEEE, June 2010.

[128] Richard Szeliski. *Computer vision: algorithms and applications*. Springer, 2010.

[129] Simon Baker and Iain Matthews. Lucas-Kanade 20 Years On: A Unifying Framework. *International Journal of Computer Vision*, 56(3):221–255, 2004.

[130] CAP5415 - Computer Vision; Lecture 18: Motion Estimation. `http://www.cs.ucf.edu/courses/cap6411/cap5415/`, 2003.

[131] Louis Le Tarnec, François Destrempes, Guy Cloutier, and Damien Garcia. A proof of convergence of the Horn-Schunck optical flow algorithm in arbitrary dimension. *SIAM Journal on Imaging Sciences*, 7(1):277–293, 2014.

[132] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2 edition, 2003.

[133] AccelerEyes. ArrayFire Examples: Optical Flow. `http://www.accelereyes.com/arrayfire/c/examples_2image_processing_2optical_flow_8cpp-example.htm`.

[134] James R. Bergen, P. Anandan, Keith J. Hanna, and Rajesh Hingorani. Hierarchical model-based motion estimation. In *ECCV '92: Proceedings of the Second European Conference on Computer Vision*, pages 237–252, London, UK, 1992. Springer-Verlag.

[135] Erin Carson and James Demmel. A residual replacement strategy for improving the maximum attainable accuracy of s-step Krylov subspace methods. *SIAM Journal on Matrix Analysis and Applications*, 35(1):22–43, 2014.

[136] Erin Carson and James Demmel. Error analysis of the s-step Lanczos method in finite precision. Technical Report UCB/EECS-2014-55, EECS Department, University of California, Berkeley, May 2014.