KFusion: Obtaining Modularity and Performance with Regards to General Purpose GPU Computing and Co-processors

by

Liam Kiemele
B.Sc., University of Victoria, 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Liam Kiemele, 2012
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

KFusion: Obtaining Modularity and Performance with Regards to General Purpose GPU Computing and Co-processors

by

Liam Kiemele
B.Sc., University of Victoria, 2011

Supervisory Committee

Dr. Yvonne Coady, Co-Supervisor
(Department of Computer Science)

Dr. Aaron Gulliver, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Yvonne Coady, Co-Supervisor
(Department of Computer Science)

Dr. Aaron Gulliver, Co-Supervisor
(Department of Computer Science)

ABSTRACT

Concurrency has recently come to the forefront of computing as multi-core processors become more and more common. General purpose graphics processing unit computing brings with them new language support for dealing with co-processor environments such as OpenCL and CUDA. Programming language support for multi-core architectures introduces a fundamentally new mechanism for modularity—a *kernel*.

Developers attempting to leverage these mechanism to separate concerns often incur unanticipated performance penalties. My proposed solution aims to preserve the benefits of kernel boundaries for modularity, while at the same time eliminate these inherent costs at compile time and execution

KFusion is a prototype tool for transforming programs written in OpenCL to make them more efficient. By leveraging loop fusion and deforestation, it can eliminate the costs associated with compositions of kernels that share data. Case studies show that Kfusion can address key memory bandwidth and latency bottlenecks and result in substantial performance improvements.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Yvonne Coady, for mentoring, support, encouragement, and patience.

Angela Bello, for not only putting up with but supporting me for so long

Donna Long, for being in the office across the gap and sharing many a pot of coffee

Natural Science and Engineering Research Council of Canada, for funding me with a CGSM.

University of Victoria, for funding me with a Fellowship.

If history is to change, let it change. If the world is to be destroyed, so be it. If my fate is to die, I must simply laugh.
Magus (Chrono Trigger)

Chapter 1

Introduction

The nature of computation is changing. Multi-core systems are now ubiquitous. In order to see performance increases, developers need to take advantage of concurrency and parallelism. Co-processors are also becoming more and more common and affordable. They are now viable alternatives which can be used to offload large amounts of computation. General Purpose GPU computing has effectively turned commodity graphics processing units into highly parallel floating point co-processors. Graphics processing units can be leveraged to accomplish a large amount of computation in parallel. Other architectures such as the IBM cell processor, and Intel's Xenon Phi also have elements which operate as co-processors. In high performance computing, GPGPU techniques are becoming more common and clusters are now being outfitted with graphics cards[1].

Using co-processors can provide a significant performance increase through parallelism, but also brings challenges associated with memory and bandwidth. In this environment, obtaining performance in parallel systems becomes a combined challenge of using both computational and memory resources effectively. Unfortunately it is unclear how optimization practices align with the basic tenants of software engineering—such as modularity. Often performance optimizations cause a breakdown in modularity. Breaking down modularity by combining modules often creates opportunities to improve data reuse and leverage memory and bandwidth more efficiently. This makes performance and modularity opposing implementation and design decisions.

This compromise is unacceptable. Modularity is more or less required to ensure complex systems can be reasonably designed, developed and maintained. Even small systems benefit from the ability to reuse code. When developing high performance

code, elements are likely to change and involve complex implementations. This requires modularity as it becomes paramount to avoid cascading changes and to isolate difficult design decisions. As systems are migrated to different platforms, changes in the optimizations may be required and it is beneficial to contain these changes to modules. High performance systems require modularity in order to be implemented efficiently, but unfortunately modularity may inhibit performance.

Developers need to be able to obtain performance while maintaining modularity. This work proposes Kfusion: a source to source transformer for OpenCL designed to bridge the gap between modularity and performance. OpenCL is portable language which covers both GPGPU computing and standard parallel execution on a standard CPU. Through code transformation, Kfusion can take modular OpenCL code and create monolithic performance kernels. Performance is increased through three optimizations: loop fusion, deforestation and asynchronous communication. In the best case scenarios explored, KFusion increased performance by approximately 4 times. In the worst case explored KFusion did not degrade performance. KFusion should almost never degrade performance as it transforms existing code and does not generate new code.

Kfusion is unique in that it is a low level transformer which operates based on a high level overview. This allows for the application developer to designate transformation operations based on high level concepts such as dataflow, while maintaining the low level high performance semantics. Other popular approaches to this problem have used a code generation approach by matching and replacing various section of standard C code. Transformations does not preclude search/replace and further maintains domain specific optimizations.

The remainder of this chapter details the contents of this thesis. Chapter 2 details background and related work. In this I discuss the founding works in modularity and concurrency followed by a discussion of GPU architecture and how these platforms are programmed using the OpenCL language. This section concludes with a discussion of related works involving other OpenCL code generators or optimization compilers.

Chapter 3 details the Kfusion source to source transformer. It explains how the tool operates and fuses OpenCL kernels to produce monolithic performance code. This section covers the required annotations used by Kfusion as well as the fusion process in detail. An example problem is given in order to provide the user with an idea of the fusion process.

Chapter 4 details the series of case studies undertaken to show how Kfusion im-

proves performance. I investigate three cases: image manipulation, linear algebra and a small physics simulation. Image manipulation provides a very data intensive example and the best performance case. Linear algebra provides an example of real world applicability as well as easily verifiable test cases. The physics simulation shows how one can improve a small section of OpenCL code in a larger application with additional components.

Chapter 5 is the analysis of the tool in terms of performance and usability. I do a qualitative analysis of Kfusion's usability compared to manually fusing the code. This is presented in terms of the number of lines of code Kfusion generates and how few annotations are required to convert a given set of functions into Kfusion compatible code. I also discus the software engineering trade-offs present within Kfusion. Quantitatively I examine the performance increase obtained from Kfusion for each case study.

Chapter 6 provides conclusions and future work. Primarily I discuss the successes of Kfusion as a stand alone tool as well as the current weaknesses and limitations present. Further work with regards to improving Kfusion further is discussed as well as areas in which Kfusion could be used to further improve the state of the art.

Chapter 2

Background and Related Work

This chapter will further explore the motivation for this work. First it covers previous work on modularity and concurrency and why both are important in modern systems. Then there is a brief overview of GPGPU computing as well as the relation of GPUs to other notable architectures. A discussion of the major components of the OpenCL programming language is followed by related work using OpenCL to produce high performance code. This covers both code generators and optimizers for the OpenCL language as well as technologies that could be related to the future work of this project. Finally there is a brief discussion of what differentiates the KFusion transformer previous works. Kfusion is discussed in detail in Chapter 3.

2.1 Modularity

The software engineering community has continued to demonstrate that modularity is key to developing quality software. Initial work in modularity includes Dijkstra's *THE multiprogramming system* [14] and introduced a layer based modularity. Upper layers could only depend on lower layers and this protected the rest of the program from implementation complexities. This work was partially driven by the need to overlap I/O with computation in order to improve resource utilization.

Parnas' initial work in modularity set the foundation for decomposing a system into modules [39]. He reasoned that a system should be broken down by design decisions which were likely to change as oppose to execution path. He also reasoned that this separation should be enforced by information hiding. These criteria along with information hiding were designed to assist with software engineering challenges

and prevent changes cascading through a given system. Later Dijkstra continued along these lines, suggesting the model for developing programs in the 1970s was fundamentally broken [15].

Support for information hiding and data abstraction began to be supported in languages such as Simula [12] and CLU [32]. These programs allows for operations to modify the state of the data, while keeping the internal structures hidden. This way, a programmer can use a module without ever knowing the internal representation.

Object oriented programming has since become a dominant paradigm in software design for several reasons. Modular software is much easier to design and maintain. A system can be broken down into components and each one can be designed and implemented independently. Each module can be assembled into a complete system. If a given component needs to be changed, it is isolated from the system and therefore can be altered without effecting the entire system. Indeed modules can be replaced with completely different implementations as long as they maintain the same interfaces. Object oriented programming is part of many major programming languages such as C++ [47], C# [17] and Java [31]. They cover and wide variety of platforms and are used extensively to implement a wide variety of systems.

Aspect-oriented programming [24] established that with the right language constructs, the benefits of modularity can be extended to include concerns that inherently span several modules in a system. As aspects, these *crosscutting* concerns are no longer tangled within the codebase, and instead can be modularized using language extensions. One area aspect oriented research has been touched upon, but not sufficiently is performance. While one case study in 2011 [33] showed that aspect oriented programming did not cause a significant performance hit, few studies have looked into how to improve performance using aspects.

2.2 Concurrency

In 2005, Herb Stutter put forth a paper *The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software* [49]. This work notes why we need a fundamental change toward concurrency. For the longest time the clock speed of processors continued to increase and operations became more and more optimized. Unfortunately this trend is over and we cannot expect CPU speeds to increase significantly. What is increasing is the number of processors available and now even commodity systems are multi-core. This has pushed concurrency into the forefront.

In order to continue to see performance gains in-line with hardware, software needs to become parallel. There are a multitude of ways to achieve concurrency. They can be generally broken down into two categories: task parallelism and data parallelism and while they are not mutually exclusive they are different approaches [48]. Task based parallelism involves executing several different tasks at once. Data based parallelism involves breaking up a single task into many parallel computations based on data. This can be considered SPMD parallelism: single process multiple data.

Significant work has been done to refine those two broad categories into more specific yet general techniques which can be applied to many parallel problems. This includes pattern languages which can provide general solutions to wide variety of parallel problems. Such languages include the *Our Pattern Language*(*OPL*)[34] and *Pattern Library for Parallel Programming*(*PLPP*)[35] developed by the Berkley par-lab including Tim Mattson. [22, 36]. The OPL is designed to allow for developers to navigate both high level and low level design decisions and focuses on forces which effect design decisions. While they often focus on trade offs with potential implementations, they do not always focus on resulting software—which could harm modularity.

There currently exist several technologies and languages to achieve parallelism. At a high level, one can use multiple processes operating in parallel and communicating via message passing through standards such as MPI [51]. Each process can execute concurrently on the available hardware. This has advantages and disadvantages. The main benefit is each process has a separate memory space. This prevents data conflicts and allows for the task to be distributed. MPI is commonly used to achieve concurrency in clusters and supports point to point and as well as collective communication operations. Synchronization is typically achieved through the nature of message passing calls. For instance an operation may block until a message is sent or received. MPI also supports barriers which force all processes to reach a certain point before continuing.

At a lower level there are many thread based technologies which allow for concurrency within the same address space. This includes basic technologies such as Pthreads [29]. Pthreads are typically used to fork execution by executing functions in parallel. Another interesting technology in this space is OpenMP [11] which allows for parallelization through the insertion of single line pragmas into serial code. There also exists various thread pooling libraries which attempt to mitigate the performance overhead caused by launching a thread by maintaining a thread pool to which work is assigned. The main advantage of a thread based approach is the reduced overhead

and shared memory space. Shared memory does create contention, but also allows for high performance systems as no communication is required. Using threads as oppose to processes is a much more lightweight solution than using processes. While we cannot achieve distributed computing with threads, threads are often used to achieve parallelism within a given a single system or a node of a larger cluster.

Finally at an even lower level there is SIMD parallelism (Single Instruction Multiple Data) which is entirely rooted in data level parallelism. While others forms of parallelism attempt to execute more instructions, SIMD attempts to do more work with a single instruction. This typically involves mathematical operations in which several values are concurrently loaded from memory, operated on and then stored in memory. This is also often known as vectorization. The number of values operated on in parallel typical coincides with the catch width of the hardware. This means that most SIMD instructions are limited to hardware specific intrinsics and there are very few widespread portable implementations.

There are also other avenues which are used to achieve parallelism such as heterogeneous computing. OpenCL [46] is an industry standard allowing for data parallelism. A unique feature of OpenCL is that is it is portable to a wide variety of hardware. This includes GPGPU programming which leverages graphics processing units to achieve performance. This is often beneficial because a given GPU can have several thousand cores capable of processing a staggering number of SIMD instruction in parallel. Unfortunately bandwidth and latency can prove harmful in terms of performance as data must be transferred over a relatively slow PCI bus.

Leveraging parallelism allows us to maintain modularity in most—if not all—cases. For instance, OpenMP requires no changes to the original code and there exists object oriented frameworks which leverage OpenCL. This includes building object oriented libraries such as ViennaCL [44] and initiatives such as Copperhead [8]. Copperhead allows for a high level code to be compiled into CUDA kernels and then run on GPUs.

Concerns with regards to concurrency that span the system. Primarily we need to use the hardware to its full capacity. This has led to design patterns which span levels of parallelism such as *Three Layer Cake* [42] by Robison et al. Three layer cake allows us to combine all three levels of parallelism previously mention: message passing, fork-join and SIMD. An image representing the three forms of parallelism can be seen in Figure 2.1. This could make it difficult to build a modular system as the types of parallelism run into each other. Primarily on how does one divide work between thread level parallelism and instruction level parallelism.

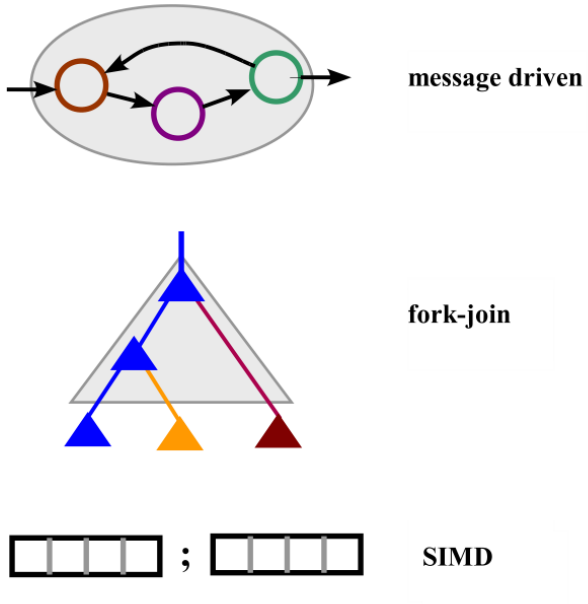

Figure 2.1: Three Layer Cake Pattern as presenting by Robison et al.[42]

2.3 General Purpose GPU Computing

General Purpose GPU Computing gives us the ability to use graphics processing units for general computation. This is extremely beneficial. While a modern commodity CPU may have approximately 2-8 cores with a upper limit of about 16, a GPU can have over 3000 stream processors [2] which is equivalent to approximately 92 conventional cores. They also have access to SIMD instructions which increase throughput. Together these features allow for a high level of parallelism which we most likely could not achieve otherwise on commodity systems.

GPUs are also optimized for floating point operations as well as provide efficient hardware implementations of many operations. GPU computing is relatively new, but many major hardware vendors, such as AMD and Nvidia, have provided SDKs and guides for development[38, 4]. This section will go into detail about GPU and programming and what makes it different from a standard multi-core environment.

2.3.1 Parallelism

GPUs have a significant increase in cores and they can be used together to create a high level of parallelism. As mentioned previously a single GPU can have a large number of cores capable of vector instructions. As such this leads to SPMD (single process, multiple data) and SIMD (single instruction, multiple data) parallelism. SPMD parallelism deals with executing instructions concurrency. This involves breaking a problem down based on data in order for many operations to execute in parallel. The most common example of this is a parallel *for* loop . SIMD parallelism is about getting more work done for each instruction. This involves leveraging single instruction which accomplish the work of several. A simple example of this is a vectorized *add4* operation will accomplish 4 *add* operations with a single instruction.

SPMD: Data Parallel Execution

Each core on a GPU executes the same set of instructions. Most code on a GPU is effectively the internal of a *foreach* loop with each core executing one iteration of the loop. It is possible to execute thousands of iterations in parallel. This can produce a significant performance increase, but these cores comes with two major restrictions:

1. Each core is part of a *wavefront* group of cores (typically of size 32 or 64).

2. Every core in a group must execute the same instructions at the same time. Effectively the operate in lockstep of each other.

This is due to hardware restrictions: groups of processor effectively share registers and memory and GPUs are designed to be *stream* processors and execute the same operations on all input data.

These two restrictions make the programming model slightly different. Control flow structures can cause serious performance degradation. For instance, should a GPU code contain an *if-else* statement which is not executed exactly the same by every core in the same group, this will cause a divergent branch. The statement will have to be executed twice serially—once for the cores which entered the *if*, and once for the cores entering the *else*. This can greatly reduce the available parallelism. This sort of problem can also occur for irregular loops. Some of these performance hits can be avoided with techniques such as masking. This makes GPUs amenable to some operations more than others. Dense matrices for instance have extremely regular operations where each core will do the same instructions, sparse matrices—depending on the storage format—may cause a prohibitive amount of divergent branching.

Each core is also capable of sharing data with nearby cores, but not all cores. Another form of available parallelism can be found in vectorization.

SIMD: Vectorization

Each core is capable of executing various vector instructions. This further improves performance as a single instruction can complete the work of several. Depending on the device, a typically SIMD instruction will accomplish two to sixteen operations in the space of one. A typical vector size is four.

Vectorization provides another method to use bandwidth effectively. When a piece of data is loaded into memory, its adjacent values will also be loaded. Memory access is aligned properly, loading one value will automatically load the next several required. Using vectorized instructions allows the hardware to really take advantage of this. Several values will be loaded, operated on and stored and require a minimum amount of I/O. Using bandwidth effectively is important and vectorization provides an excellent way to do this.

2.3.2 Memory and Bandwidth Concerns

When moving computation to the GPU, there are two major concerns: data transfer and the memory hierarchy on the GPU. In order to obtain performance, both must be used effectively.

Data Transfer

Data transfer is relatively straightforward, but can be costly. Data must be explicitly moved to and from the GPU during execution. While the initial latency associated with the transfer may cause a performance hit, it can be compensated with by pipelining. This makes GPUs very throughput oriented and asynchronously transferring data to and from the GPU becomes important. Using pipelining it becomes possible, depending on the density of the computation, to eliminate the latency associated with the transfer.

That being said, when thinking of whether or not to move a computation to the GPU, the data transfer should be considered. An application which lends itself well to the GPU may execute several order of magnitudes faster than on the CPU, in this case the transfer time—though prohibitive—may overshadowed by the performance benefit.

Memory Architecture

GPUs have a slightly different memory architecture. The largest difference is that there is not a direct equivalent to cache. Data is loaded directly from global memory, used and then stored again. This can make access costly and repeated access to variables must be avoided. There are faster, but much smaller memory which is local to the individual cores. Accessing memory should be done consecutively in order to take advantage of coalescing which combines memory access and allows for efficient pipeline.

GPUs do have a form of faster shared memory but it must be explicitly loaded and stored. Also between kernel execution, it becomes invalid and local memory cannot be used between kernels. This will be discussed in detail during the OpenCL section of this chapter as that details the relevant abstractions.

2.3.3 Relation to Other architectures

When looking at other types of co-processors, GPUs have similarities. The first is that most co-processors exist on the other side of a bus. This is going to make data transfer very important and asynchronous data transfer becomes a must in order to ensure computation can move forward. Vectorization is also standard component of co-processors. Two similar examples are the IBM's Cell and Intel's new MIC.

IBM's cell architecture [20] has several accelerators referred to as SPUs. They are separated from the main CPU by a bus. Each SPU has two threads and supports vectorization. This effectively means a similar pipelined data transfer model is required and SPUs will leverage a similar type of parallelism and suffer from performance hits due to branching instructions.

Intel's new Xeon Phi is Intel's attempt at a co-processor [19]. It is step towards a more general purpose processor. Much like a GPU it will be inserted into a PCI slot and have similar transfer concerns, support vectorization and have a large number of cores. The Xeon Phi will have a much more CPU like architecture. It can be programmed with a variety of APIs such including OpenCL.

It is also worth considering standard CPU architectures and how well this work applies to them. CPUs have very few cores in comparison, but significant cache. Performance characteristics should be very different. My case studies discussed later in this thesis were also applied to a general purpose CPU in order to make work generalize to other systems.

Device	Cores	Memory Type	Transfer Mechanism	Remarks
Intel i7-2600k	4 cores, 2 tthreads/core	L1-L3 Cache	N/A	Standard intel CPU.
Tesla 2075	14 cores (488 stream processors)	local memory	PCI-E bus	Nvidia Coprocessor
IBM Cell	8 SPU Accelerators	local store	Interconnect Bus	Each core has two threads. Local store acts as cache but explicitly loaded
Xeon Phi	32-50 cores, 4 threads/core	L1 and L2 Cache	PCI-E Bus	Intel Coprocessor currently in development

Table 2.1: A quick overview of various platforms to demonstrate how they relate. The first two processors will be used in this work to give an idea of how Kfusion operates on different platforms. The following two give an idea how other architectures are similar. Data transfer will involve a significant cost as it must traverse a bus and the caching mechanisms may be nonstandard. Both applications are parallel, so it becomes important to minimize I/O an latency in order to take full advantage of the available hardware.

2.4 OpenCL

OpenCL is a newly developed industry standard managed by the Khronos Group [23]. It aims to provide a single standard for programming a wide range of devices: CPUs, GPUs, various accelerators and other systems. The idea follows previous standards, such as OpenGL, in order to provide a powerful interface for developers. It gives the programmer a portable abstraction with which to access the underlying hardware. OpenCL is my language for choice for this work. It is extremely portable and produces performance that is similar to CUDA. Using OpenCL allowed me to develop the codebase for GPUs, but also attain CPU results which will be discussed later in this work.

This section will cover the OpenCL execution mode, its units of functionality: the kernel, memory spaces and data-types.

2.4.1 Execution Model

OpenCL supports SIMD parallelism with the addition of portable vector instructions. Computation is accomplished by executing kernels on a target device. Its execution model separates the execution between the *host* and the *device*. The host is the general purpose hardware which acts as a platform for any number of OpenCL enabled devices. The device is the target GPU, CPU or accelerator.

Executing kernels is slightly more complicated than standard parallel programming as there are additional steps moving data to and from the device and executing the given kernels. A typical OpenCL program will have this work-flow:

1. Initiate OpenCL - Executed Once
   (a) Create an OpenCL shared memory context
   (b) Initiate the OpenCL devices
   (c) Create OpenCL queues for each device
   (d) Compile OpenCL kernels for each device
2. Transfer data from the host to the device
3. execute kernels over the data
4. Transfer data from the device to the host

Kernels and memory transfers are assigned to an OpenCL queue and are either executed in order or asynchronously based depending on the current execution settings.

2.4.2 Kernels

OpenCL is based on compute kernels which operate over a global work space. OpenCL kernels are compiled at run time for the target device. This ensures that the best compiler for the device is used. It improves compatibility as well as ensures hardware specific optimizations are performed. This does have a side effect of requiring run-time compilation which adds an additional level of complexity to a working OpenCL application.

An example of a kernel is shown in Figure 2.2. The code will sum up each of the elements in *v2* and *v1* in parallel. In this case the global work space is the vector and the code present here is executed for each set of elements in the vectors. Each output value will be computed in parallel. Effectively, each element of the vectors gets its own thread. Each instance also has a *global id* which can be used to carry out computations on the correct segment of data. In this instance the global id is stored as i, this code operates much like a parallel *for* loop.

```
__kernel void vectorAdd(__global float* v1,
global float * v2, global float * v3)
{
    int i = get_global_id(0);
    v3[i] = v2[i] + v1[i];
}
```

Figure 2.2: OpenCL Vector Addition Kernel

Kernels provide the essential building blocks for OpenCL computation and are executed on any single computational device, such as CPU or GPU. They are queued over any given global works space. A workspace can have up to three dimensions. The definition of a workspace can probably best be described as the limits of a for loop or each element of a $foreach$ loop. Two or three dimensional workspaces are similarly analogous to a series of nested for loops. Problems that lend themselves well to 2D workspaces are ones which deal with images, matrices or other 2D structures.

Work Groups

Kernels execute across a workspace and each element of the workspace also belongs to the global work group and the local work group. Work group sizes are specific to the problem being solved, but also should compliment the underlying hardware.

For instance GPUs operate best when both the local and global work groups sizes are multiples of 64. Multiplies of 64 produce an ideal mapping between the schedules work and the available hardware. The global work group encompasses the entire problem set. Each member of the global work group will execute the kernel code in parallel.

Each member of the global work group also belongs to a local work group. Local work groups can be any size which divides the global work group size equally. Essentially the global work group is composed of a set of local work groups which each contain a part of the global computation. Members of a local work group share memory and execute on the same computational unit. This becomes key for two reasons: shared memory and synchronization. Members of a local work group can share information through *local* memory. Synchronization is discussed in the next subsection.

Synchronization

OpenCL provides an event based synchronization system to allow for correctly ordered execution of kernels. Using events, we can build dependency graphs or task graphs and have OpenCL execute them in order. Each kernel when launched can be set to wait on a list of events and also signal an event when completed. This is perhaps best used when leveraging multiple devices. Within a single device, in order execution produces better results than event based synchronization.

Synchronization can also occur within kernels, but only within the same local work group. Memory fence operations can be used to ensure that all global or local memory access has been flushed. Barrier instructions require all threads to reach them before execution can continue. Generally synchronization is detrimental to performance and it is much better to do more work and have less synchronization.

2.4.3 OpenCL Memory Spaces

OpenCL uses a memory model with different levels of consistency, size and speed according to the following hierarchy [3]: *global, constant, local* and *private*. To provide context, these are briefly overviewed here, though we refer the reader to [3] for a more complete treatment of related OpenCL concepts such as work groups and threads.

Global memory is the largest. Unfortunately it is also the slowest memory and accessing values from this memory space is costly. This is similar to main

memory in traditional models. Accessing global memory incurs a significant cost as such global memory access should be kept to a minimum. Global memory accesses should be consecutive across each member of the global work group. Consecutive access will coalesce the memory access allowing for maximum load efficiency. Global memory is roughly analogous to main memory.

Constant memory is smaller than global memory, but allows for caching. This creates the opportunity for improved memory access under conditions where data can fit in constant memory and it is used more than once.

Local memory is faster than constant memory but smaller. Kernels can quickly access local memory, much like a cache. Though it can be painstaking for developers, GPU applications are typically designed to exploit this, as local memory can provide significant performance boosts relative to both global and constant memory. Local memory is shared among members of the local work group. Data can be loaded once, but used by many different kernels.

Private memory is the fastest memory and more akin to registers. Each kernel has its own private memory. It is fast, but extremely small. Overusing private memory will cause it overflow into global memory.

It is important to use these spaces correctly in order to achieve performance. Many highly tuned GPU applications explicitly manage data according to these constraints. For instance, accessing data in local memory can be an order of magnitude faster than global memory and avoids issues such as *bank conflicts* which introduce overheads and can occur when accessing global memory. When accessing global memory, it is also important to access values consecutively in order coalesce memory access and gain substantial advantages through pipelining. Incorrectly using global memory can cause memory reads and writes to become serial, which will incur a significant performance penalty.

With these memory spaces it becomes best practice to load data from global memory only once, store it in either local or private memory and then write out the result a single time. Local memory should be used whenever data can be shared. Data can be asynchronously copied into local memory from global memory and this can significantly improve performance as well as act as a prefetching operation.

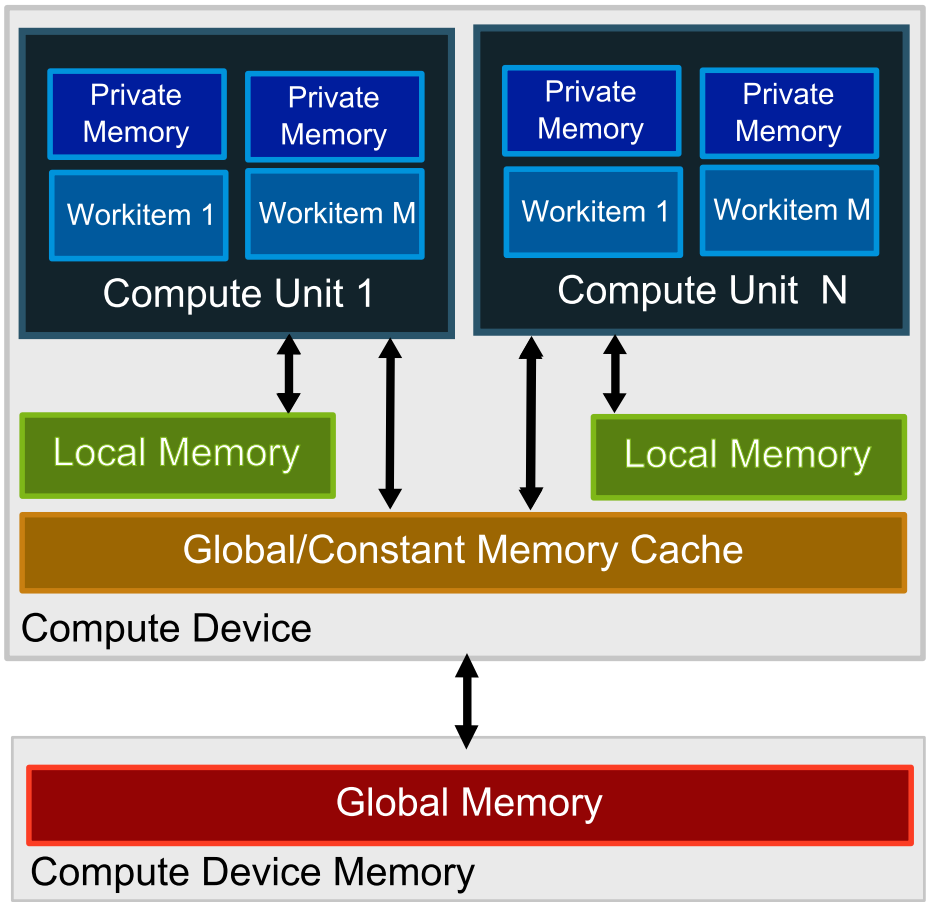

Figure 2.3: OpenCL memory Model[3]. This shows how data is moved between memory spaces. All data starts in global memory and is moved to moved to smaller, yet must faster memory spaces on the actual hardware. This includes local memory, which is as a shared memory space and private memory which is very similar to registers.

2.4.4 OpenCL Memory Objects

OpenCL supports two memory objects: buffers and images. Buffers are flat one dimensional objects which are passed into kernels as pointers. They can be directly accessed using pointers and reside in global memory. They are generally useful for most general data storage requirement. They can be extremely large, so storing two dimensional buffers as one dimensional buffers is not out the question.

Images are two or three dimensional objects which cannot be accessed directly. There are specific functions which read and write images. An image must be defined as *read only* or *write only* meaning a kernel cannot both read and write to the same image. This comes with several benefits. Images can be present in texture cache—which is extremely fast—and are stored with two or three dimensional locality. They are also cached. This means accessing nearby points on an image can be extremely fast. The main downside is the inability to read and write to the same image. This can create intermediate products.

2.5 Loop Fusion and Deforestation

This work leverages two major optimizations: loop fusion and deforestation, which are often automated in terms of their application. Loop fusion is used in imperative as well as parallel applications to improve memory locality [21, 45]. Deforestation involves the transformation of code to eliminate intermediate data structures [50, 10]. Fusion often results in deforestation, as combining operations on traversals of data structures typically removes the need for intermediate results. Both of these transformations are used within the high performance computing community in order to improve performance [5].

2.5.1 Loop Fusion

Loop fusion is a technique more common in functional languages, but has recently migrated to the field of concurrency [21]. The basic idea is simple: *for* loops are combined in order to improve data locality and parallelism.

Combining loops improves parallelism because it reduces how often threads need to be branched off or assigned work. This reduces overhead. Likewise allows hardware to better pipeline instructions as now each loop iteration contains the instructions of

multiple separate loops. This is beneficial as instruction caches can be leveraged and the latency which occurs when fetching instructions can be hidden.

Memory access is also improved. Each value will be loaded once, operated on several times and then stored as oppose to if each loop is separate in which case data may be loaded and stored several times. This allows for the use improved use of available data cache space. Data locality is improved and threads will have to wait on I/O much less. Likewise this improves the hardware's ability to prefetch the next set of required data and also improves latency.

2.5.2 Deforestation

Deforestation involves transforming programs to eliminate unnecessary trees [50]. In the case of this work, this idea can be applied to any and all unnecessary data structures and temporary variables that may occur when doing a series of operations. This generally involves re-ordering or combining operations in order to remove unnecessary values.

The benefits from this come primarily from the reduced memory footprint. If an application needs to access less memory, this will involve reduced data movement and improved memory locality. This complements loop fusion as relevant data can be kept in cache longer—significantly improving performance. Cutting out intermediate data structures will also remove the operations required to create, manipulate and delete them. This will improve instruction cache and reduce other overheads.

2.6 Optimizing Libraries and Compilers For GPGPU Programming

Managing kernels, memory spaces and data transfers to and from the target device can be difficult. In order to help with this, there has been a significant amount of related work covering the generation and optimization of GPU code.

The work in this area has primarily been concerned with generating GPGPU code from standard language such as C or Java or other frameworks. One approach converts affine programs written in C to CUDA code[6]. Affine programs are programs which perform affine transformations on data. An affine transformation can be defined as an operation on a matrix or vector which preserves straight lines. Their approach uses a polyhedral model combined with an abstract syntax tree generated from the

code to analyze dependencies and build high performance CUDA code. They are able to implement a few key operations such as memory tiling and take advantage of the GPU memory hierarchy to minimize costly I/O. One key disadvantage of this approach is it only works with affine programs.

Another approach maps OpenMP pragmas into CUDA [26]. This works well as OpenMP pragmas are relatively easy to use and many of their supported parallel constructs, such as *for* loops, map particular well to a GPU. They do several optimizations while mapping from one domain to the other. They focus primarily on global memory access and include such operations as coalescing memory access, collapsing irregular loops in order to eliminate irregular memory access, caching reused global data and reducing memory transfer to and from the host device. Their performance was initially poor, but improved through the before mentioned optimizations. The drawback with this setup is that while it may construct fairly optimized CUDA code taking care of various operations, it will not be able to take advantage of optimizations which may exist between operations. This is what my works attempts to address.

BONES is another source to source compiler targeted at transforming C to CUDA and OpenCL [37]. This is done in a similar manner to the OpenMP to Cuda as it finds areas of the source code which can be mapped to code skeletons. These are highly parameterized general parallel routines. BONES and other skeleton coding systems implement typical parallel operation such as map and reduce, but allow them to be expanded. There are other technologies in this field. They have various downsides: inserting of a large number of pragmas, requiring you to rewrite your code, requiring the use of special data structures, or only operating on affine loop transformations. BONES avoids these problems and produces code which performs close to hand optimized kernels. It also outputs intermediate code which can be tweaked by an experienced user in order to further increase performance. This is a fairly unique feature of their program. SkePu is another algorithmic skeleton approach. The key optimization they do, that Bones does not, is lazy copying of data to and from the device[13]. Lazy copying ensure data is moved a minimum number of times and only when needed, but will not allow for asynchronous copying of data which can be used to hide latency. Predictively moving data to and from the GPU several operations before it is needed could further improve performance.

The field of parallel compilers [25, 7] allows for the automatic parallelization of serial code. A typical example is the automatic parallelization of *for* loops. Generally

there are some restrictions on what can be transformed and this requires some form of static or dynamic analysis. There has been some work in the field of parallel compilers to bring automatic parallelization to GPU computing by Leung et al.[28]. They accomplished the automatic parallelization of Java code and automatically offload computation to the GPU—if it provides an increase in performance. Much like with the skeleton algorithm approach, affine program conversion and converting OpenMP pragmas into OpenCL, automatic parallelization will be able to handle specific constructs and loops but be hindered by modularity concerns. Specifically my work takes advantage of two modules executing consecutively in order to increase performance, other approaches currently do not.

The Delite framework is a very different approach compared to the previous [9]. Delite is based off of Scala which is typically used to build domain specific languages and it provides a framework for quickly building optimized domain specific languages. DSLs restrict what the programmer can do and then the compiler can perform greater levels of optimization. These DSLs can then be compiled down into Scala, C++ and OpenCL. This high level approach to low level code generation allows for a series of optimizations which span standard compiler optimizations, such as dead code elimination, to replacing high order functions with first order functions, precomputation of values, and operator fusion. Fusion touches on what my work does and essentially allows you to combine operations in order to take advantage various optimizations such as data locality. Delite provides a very high level approach to generating efficient code at compile time. This provides benefits, but in the end produces a domain specific language which is not as general purpose as my work.

Work has been done to further optimize GPGPU code in an automatic manner by Yang et al. [53]. Their work takes naive GPU kernels and optimizes them in two major respects: memory use and parallelism. The naive kernel is optimized in terms of memory access patterns, memory coalescing, vectorization and loop unrolling. They achieve performance close to, and in some cases superior to, finely tuned hand optimized code. This is an important work. It shows that kernels themselves can be optimized and improved automatically, but it does not handle optimization which may exist between kernel. My work could be leveraged in tandem with this technique to produce monolithic high performance kernels from modular components.

SPIRAL is a domain specific language for signal processing [40]. Spiral expands on the concepts of active libraries to provide a more general framework. It works much like Fastest Fourier Transform in the West [16] by generating code snippets which

are tested at compile time. The best performing snippets are combined together to make a performant implementation. While this does not directly relate to GPGPU architectures, it could most certainty be applied to OpenCL and CUDA. My work could provide the glue required to efficiently build active libraries on GPGPUs.

Framework	Type	Pros	Cons
Affine Mappings	Generator	• polyhedral model handles dependencies • loop fusion and other optimizations	• limited to affine programs
BONES	Skeleton Algorithm	• uses skeleton mappings to convert C to OpenCL • performs loop fusion	• currently limited to basic algorithms • does not handle abstraction
SkePu	Skeleton Algorithm	• same benefits as BONES • supports lazy data transfers	• must have mapping from CPU code to GPU code • does not support abstraction
Java to GPU	Parallel Compiler	• performs dynamic analysis of operations • converts Java to GPGPU code when beneficial	• Is not capable of fusing related operations • Dynamic analysis may be unpredictable
OpenMP to Cuda	Parallel Compiler	• Converts OpenMP pragma's to GPGPU Code • Works well with few additions	• Will not be able to take advantage of abstraction • does not perform fusion optimizations
DELITE	DSL Generation	• supports operator fusion • bring high level optimizations to low level	• Restricts programmer to perform optimization • DSL's are not general case
SPIRAL	Active Libraries	• brings active libraries to Signal Processing • combines performant solutions based on hardware	• not directly related to GPGPU programming
Yang et Al.	Optimizing Compiler	• optimizes naive kernels • performs close to hand optimizations	• cannot take advantage of optimization between kernels

Table 2.2: A quick overview of various platforms to demonstrate how they relate. The first two processors will be used in this work to give an idea of how Kfusion operates on different platforms. The following two give an idea how other architectures are similar. Data transfer will involve a significant cost as it must traverse a bus and the caching mechanisms may be nonstandard. Both applications are parallel, so it becomes important to minimize I/O an latency in order to take full advantage of the available hardware.

Looking across code generating tool-sets they all have a few commonalities:

- They are fairly specific. They convert a subset of a given language into a parallel implementation or generate a domain specific language which effectively does the same thing. They have a somewhat limited scope and generally tackle already parallel applications.

- They do not take advantage of inter-modular performance optimization. Most of the these frameworks cannot handle function calls. The exception is the Delite framework. Either way, we are going to be limited in our modularity when it comes to performance.

- They all achieve similar to hand tuned performance. This shows that we can effectively optimize a single, parallel, module. This does not show that it is possible to optimize combined kernels though. This is where my work comes in.

The key difference in my work is that it can leverage modularity to perform optimizations which require knowledge of the larger picture. While other frameworks convert small sections of code—typically for loops—into parallel code implementations, my work endeavors to traverse boundaries in modularity to improve existing OpenCL libraries at compile time. In this way the natural abstractions used in modular programming become tools allowing the developer to compose modular kernels into monolithic performance kernels depending on their specific use.

That being said, this work is not in opposition to the techniques mentioned previously here, but instead could complement them. A parallelizing compiler or skeleton algorithm approach could be used to create an OpenCL enabled library and my proposed tool in the next chapter could be used for further optimization and improvement. Finally existing optimizing compilers could be leveraged before or after my proposed tool in order to all a developer to create naive kernels which can be composed into high performance kernels .

It is important to note that loop fusion is used by a two frameworks present here: BONES and Delite. They fuse the high-level codebase and then produce OpenCL from the result. My work will complement this approach, as it instead applies this technique to fuse OpenCL kernels directly, in an OpenCL-to-OpenCL transformation. This has the additional benefit of deforestation to remove intermediate data structures that might result from kernel compositions, as well as redundant computation.

Chapter 3

The New Approach and Solution

When moving to general purpose GPU computing we're presented with a fundamentally different platform which is highly parallel yet suffers from memory, bandwidth and latency concerns. While memory and bandwidth may not be any more constrained than on general purpose hardware, it becomes much more noticeable with the addition of hundreds of cores. Instead of utilizing functions, GPU languages require the use of kernels, which each execute independently with a clean slate in terms of memory. In effect, the cache is reset. This makes a kernel a different modular unit as opposed to a function.

Optimizations associated with memory access patterns are hard to modularize, in particular because these patterns are often dictated by application-specific needs. For example, consider a library of operations that each modify the same data structure. Inherently, in OpenCL, this library would support these individual operations through separate kernels (and in some operations, potentially several kernels). This modularity is important as it ensures the library can be used in a general case, unfortunately modularity also hinders efficient memory access—especially due to the fact cache is reset between kernels. Though application-specific combinations of these operations might be able to amortize costs through fusion and deforestation—substantially improving performance—this information is only available at the time the application is compiled.

In this chapter I present KFusion: a novel transformation tool capable of greatly improving GPGPU performance by combing kernels at compile time. Using simple annotations, it can re-modularize kernels to improve performance. In the following section, I will discuss a motivating example for KFusion followed by a brief overview of KFusion. Then this chapter will delve into the details behind its overall design and

the transformation process.

It is important to note that this model is applicable to other forms of hardware. Most co-processors will have similar concerns. This includes platforms such as the IBM cell processor. Likewise this model is somewhat applicable to general purpose CPU hardware present in most commodity systems—should the problem involve large amounts of data as hardware and software prefetching may not be able to compensate. In order to see how it operates on more conventional hardware, the case studies presented in the next chapter were also executed on a CPU.

3.1 Motivation

This section will motivate KFusion with a short example illustrating the costs of operations on GPUs. Suppose there is the algorithm as shown in Algorithm 1 with each operation implemented by a linear algebra library. Here they are several mathematical operations which occur on vectors.

Algorithm 1 A simple mathematical example which will be used in describing the various components of the transformer. Each variable *x,y,c* represents a vector or array of length n

1: square(x) - square the values of array x
2: square(y) - square the values of array y
3: add(c,x,y) - add x and y and store in c
4: sqrt(c) - obtain the square root of each value in c

Each operation is computed by a separate kernel, which is executed in order. This modular implementation ensures each kernel can be developed and debugged individually. Modularity ensures separation of concerns and that each kernel can be reused and apply generally. An API protects the application developer using this library from having to deal with the lower level implementation details, each kernel can be optimized individually.

3.1.1 Costs of Modular Implementation

It is possible to reason about the costs both in terms of computation and memory accesses. Equation 3.1 provides a simplified formula for obtaining a performance estimation: C is the number of computation instructions and α is the average number of clock cycles required to execute each instruction. $M_G and M_L$ are the number of

memory operations at each level: global and local. $\beta_G, \beta_C, \beta_L and \beta_P$ represent the costs. These will change depending on hardware, we will assume a standard GPU. There exists more complex models [41], but this works for our motivating example.

$$T(c) = C\alpha + M_G\beta_G + M_C\beta_C + M_L\beta_L + M_P\beta_P \tag{3.1}$$

Using the Nvidia optimization guidelines, the costs become apparent and can be expressed in terms of clock cycles. The minimum latency of a global memory optimization is 400 clock cycles. Local memory on the other hand has a latency of 5 clock cycles and is considered negligible. Private memory is equivalent to registers and adds no additional clock cycles to an instruction. The maximum number of clock cycles required for non memory operation is 4. The equation then becomes simplified to be Equation 3.2. This does not include constant memory.

$$T(c) = 4C + 400M_G + 5M_L \tag{3.2}$$

A summary of the global load and store operations themselves can be seen in Table 3.1. This shows a major problem: continually loading and storing data causes a major increase in latency and consequently the number of clock cycles required to execute the kernel. Some of this can be hidden by overlapping communication and computation, but this requires a ration of approximately 100 mathematical operations for each load or store operation. This is unlikely. The OpenCL implementation can attempt to hide the latency by running a series of threads on the same hardware and context switching but this has limits depending on a series of factors. As each kernel only executes a single instruction for two loads and stores, 200 parallel kernel invocations on each core will be required to hide the latency.

Kernel	**Arithmetic Operations**	**Global Memory**	**Cost (cycles)**
square(x)	1	1 load and 1 store	804
square(y)	1	1 load and 1 store	804
add(c,x,y)	1	2 loads and 1 store	804
sqrt(c)	1	1 load and 1 store	804
total	**4**	**9**	**3216**

Table 3.1: List of kernels and the load and store operations present. This gives us a general idea of a major performance indicator. The case of the add instruction, some latency can hidden by overlapping the load instruction.

3.1.2 Improving Performance by Breaking Down Modularity

Suppose there is a new kernel which accomplishes all the individual kernels functionality. This new kernel is referred to as: *square_square_add_sqrt(c,x,y)*. It removes any unnecessary results and only computes the final output. As such, only a minimum number of loads and store operations are required. The amortized costs can be seen in Table 3.2.

Kernel	**Arithmetic Operations**	**Global Memory**	**Cost (cycles)**
square(x)	1	1 load	404
square(y)	1	1 load	404
add(c,x,y)	1	0	4
sqrt(c)	1	1 store	404
total	**4**	**3**	**1216**

Table 3.2: Fused Kernel Costs: List of kernels and the load and store operations present. This reduces the load and store operations to 3 and reduces the cost to approximately a third of previous.

The end result is a reduced number of cycles required to execute the kernel—to about a third. Also the computational density is increased. As oppose to 4 mathematical operations to 9 memory operation, there is now 4 arithmetic instructions for 3 memory operations. This will improve OpenCL's ability to hide latency by overlapping communication and computation. The results of creating a monolithic kernel can be seen in Figure 3.1.

The key concept is data flow. If operations are combined, data is reused while still in cache or ideally hardware registers. This is a simple example, but it can be applied to many other types of problems, such as image processing, as long as there is shared data. Instead of repeatedly loading and storing a value into and out of memory, it is loaded once, operated on using as many operations as required and then stored. This transforms a set of functions into streaming operations.

Unfortunately creating mono-kernels causes a breakdown in modularity. As systems increase in size and complexity, manually fusing functions to obtain better performance quickly becomes unsustainable. This also suggests the idea that a library user has the ability to fuse functions and this breaks any concept of an API. The implementation details are no longer separated from the high level interface. This presents a major software engineering problem.

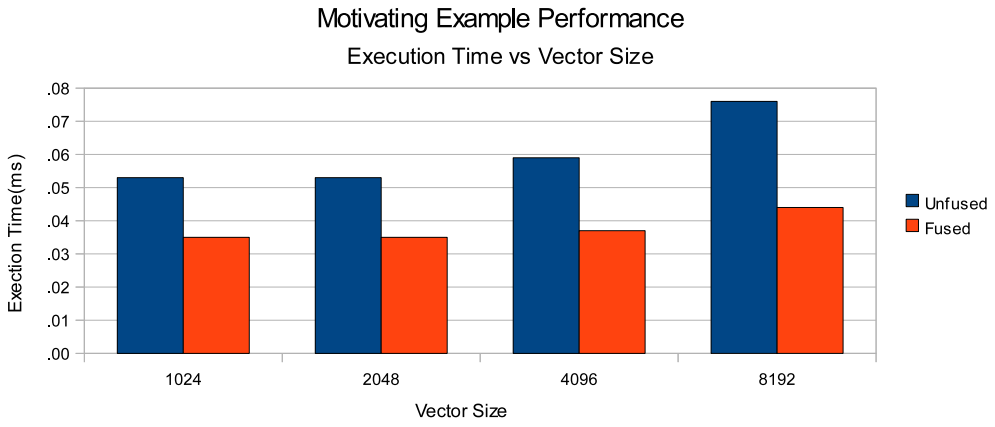

Figure 3.1: A performance comparison between unfused and fused kernels. There is a significant gain primarily do the amortization is memory access costs. In both cases, OpenCL can mitigate some latency associated with memory access, but fusion greatly helps with this. It's worth noting that the monolithic implementation's performance benefits scale well as the vector size increases.

3.2 Proposed Tool: KFusion

There are clear benefits to modularity, but in many cases systems must also have high performance. Both modularity and performance could be considered hard requirements for a large portion of systems. Unfortunately, there has almost always been conflict between modularity and performance and this is no different. As such this work attempts to lay the foundation for fusing kernels at compile time in order to preserve modularity during development as well obtain the performance of monolithic code during execution.

To explore this idea, I introduce a prototype tool, KFusion, which is an OpenCL-to-OpenCL transformer. KFusion uses a semi-automated approach to accomplish low-level kernel fusion that maintains original source semantics. This allows for independent, low-level, kernels to be transformed into high-performance monolithic kernels on an application-specific basis.

At a high level, KFusion takes a set of annotated library functions along with set of OpenCL kernels and performs an analysis of each kernels inputs, outputs and synchronization requirements. The annotations themselves are simple and require only the addition of a few lines of code interspersed with the existing application. Then it examines the uses of the library function in the main application and creates new functions and kernels which better match the given use case. These new kernels are fused versions of the originals. Fusing allows us to eliminate I/O, intermediate products and move computation around to take advantage of asynchronous communication. This effectively custom tailors a library at compile time to the specific use case of its user.

One area where this work differs is that while other works may convert regions of C/C++ code into OpenCL, this does not inherently support or benefit from modularity. My work leverages modularity to allow the programmer to see the larger picture and how data flows from one operation to another. KFusion can use the high level dataflow to inform low level optimization and this type of optimization supports and benefits from good software engineering practices. It finds the optimizations which exist between modules, while maintaining the software engineering benefits of using them.

3.3 KFusion Design

KFusion takes modular OpenCL kernels and the libraries which use them and create new kernels and library function at compile time. This allow an application developers to attain low level performance benefits without breaking the libraries API. At this point, I would like to explicitly state that KFusion is used by two different developers:

Application Developer - Uses the KFusion enabled library to develop a given application. They add annotations to the application files and specify which regions of the code should be fused.

Library Developer - Builds the KFusion enabled library and kernels associated with it. They add annotations to the library files and kernel files.

The distinction being the library developer handles the low level details associated with fusion while the application developer can naively attempt to fuse them with the addition of a few annotations. The annotations are added to an otherwise complete library or application.

KFusion is designed to operate with three different sets files covering the application, library and kernels:

The application files accomplish a specific task using a given OpenCL library provided by a library developer. This can include libraries such as image manipulation, linear algebra, physics simulations and many more. The application file will have fusion regions which call several library functions consecutively.

The library files contain the implementation of operations used by the application files, through a published API. These files provide a layer of abstraction between the OpenCL specific implementation and the application it is servicing. Libraries are generally responsible for moving data, setting arguments and requesting the execution of kernels. Each library function designated to be fused must contain one OpenCL kernel.

The kernel files contain the OpenCL kernels to be executed on the GPU or other target device. These kernels are leveraged by the library function calls to accomplish work.

The semi-automated approach requires developers of kernel-based libraries to annotate library functions as well as OpenCL kernels. Library annotations provide

synchronization information detailing which functions cannot be fused. Kernel annotations provide information on how to fuse functions based on shared data.

KFusion has two major phases: (1) analysis of the code-base and (2) synthesis of new functions and kernels to improve performance according to application-specific needs. At a high level, for each kernel within a fusion region, outputs are fused with matching inputs to produce monolithic kernels. The approach is semi-automated where both the application files and kernel files require annotations to explicitly support analysis and synthesis. No input is required during the fusion.

As KFusion goes through the stages of the analysis and synthesis of new kernels and functions I will give a simple example of mathematical operations as shown in Algorithm 1.

3.3.1 Annotations

KFusion requires several annotations at various levels in the program. This subsection details them and explains how they are used. Specifically, the application annotations, library annotations and kernel annotations are discussed in detail.

Application Annotations

At the application level, only one annotation is required. It defines which functions should be fused using the KFusion transformer:

kfuse(param){ ... } - Defines a set of function to fuse using the KFusion transformer. It should surround any supported library functions which the application developer wants to be fused.

And example of the annotation is used is shown in Listing 3.1. Kfuse will attempt to combine any functions it can based on synchronization annotations detailed at the library level.

Library Function Annotations

Library functions are annotated with synchronization information through a simple *pragma* before the function definition. They determine if a function and any of the kernels it contains can be fused. Most functions will not require this annotation, but they allow for restrictions which can be used to ensure safe fusions. Example of their use can be seen in Listing 3.2

```
kfuse(x,y,c)
{
  square(x)
  square(y)
  add(c,x,y)
  sqrt(c)
}
```

Listing 3.1: The application level annotation present required to fuse the example problem. This signals to KFusion that these function should be fused. The resulting fused implementation will make the operations non-destructive for all vectors except c which will output the correct result.

```
#pragma sync out
void dotProduct(Vector *a, double &result);

#pragma sync in
void matrixMult(Vector *b, Matrix *A,
   Vector *x);
```

Listing 3.2: In example of synchronization annotations for a matrix multiplication and dot product operation. The matrix multiplication requires the entirety of x and the dot product reduces to a single value.

#pragma sync in - This function requires a synchronized input. It will not be fused with any previous functions.

#pragma sync out - This function requires a synchronized output. It will not be fused with following functions.

There may exist instances where a kernel requires synchronized input and output, these will not be fused and effectively provide breakpoints. These situations will arise when either a kernel requires the entire result of a previous operation or when a kernel ends in a reduction type operation. Another good example is a resize operation which will end in a result which will be incompatible with the following kernels. The synchronization *pragmas* are currently too course grained and could be improved by specifying which specific inputs and outputs must be synchronized. The stronger limitation do prove to be safe, but may be unnecessary.

Each function in the library file meant to operate with KFusion must contain at least one OpenCL kernel. Each function is assumed to contain one kernel. If a function has two kernels which require synchronization between them, this will currently cause problems and may inhibit correctness. Currently, it is recommended that a function which executes two OpenCL kernels should be split into two functions.

Kernel Annotations

Kernel annotations mark up the OpenCL kernels which are executed on the GPU or other target device. The annotations highlight fusion opportunities in kernels which operate on shared data as well as limit some optimizations which will occur after fusion. The annotations are as follows:

kload { ... } - Kload annotates regions of a kernel whose operations are load operations

kstore { ... } - Kstore annotates regions of a kernel whose operations are store operations

#pragma immovable - This annotation relates only to a few chosen asynchronous instructions *prefetch* and *async_work_group_copy*. KFusion will attempt to move these statements to as early as possible in the execution of a kernel. This pragma prevents that.

An example of the *kload* and *kstore* annotations can be seen in Listing 3.3, while an example of the immovable annotation can be seen in Listing 3.4. There, they detail the load and store operations.

Kload is used to outline which private or local variables are assigned to from global memory. These effectively mark out which variables are assigned values from the kernels parameters. Kload is also used to mark any variables which will be carried through and used to store data at the end of execution. Kload must occur in the root scope of a given kernel and cannot be within control structures such as *loops*. A good rule of thumb is that all initial declarations in a function are Kload instructions while temporary variables are not. Kload cannot be used to mark data within loops.

Kstore is used to annotate any output statements which assign to global memory. Generally this is a kernel parameter. A good rule of thumb is that these are the last few or single last assignment statement in a kernel. They mark the outputs. Later on in fusion, the value which assigns to the given global parameter will be used to replace a load instruction which accesses global memory.

#pragma immovable is used to annotate that the following statement should not be moved during optimization. This relates to OpenCL asynchronous instructions dealing with memory transfer and prefetching. In the event kernels are combined, any asynchronous instructions at the root scope will be moved earlier in execution in order hide latency and maximize the use of available bandwidth. In most cases, moving these functions forward in the code should not cause problems, but in the even data is transferred to and from a location multiple times, some of these statements will have to labeled as immovable.

3.4 KFusion Transformation Process

This section will detail the process used by KFusion to produce new kernels and functions and carry out several optimizations. There are two major phases: Analysis and

```
__kernel void add(__global double * c,
                  const double alpha,
                  __global double * a,
                  const double beta,
                  __global double * b) {

  kload{
    int gid = get_global_id(0);
    double aVal = a[gid];
    double bVal = b[gid];
    double cVal;
  }

  cVal = alpha*aVal + beta*bVal;

  kstore{
    c[gid] = cVal;
  }
}
```

Listing 3.3: An example of annotating load and store operations with kload and kstore

```
__kernel void immovable(__global double * A,
    __global double * x,
    __global double * b) {

  kload{
    __local double[32] var;
  event_t copy = async_work_group_copy
    (var,x,32,0);
  }
  ...
  #pragma immovable
  copy = async_work_group_copy
    (var,&x[32],32,0);

  ...
  #pragma immovable
  copy =  async_work_group_copy
    (var,&x[64],32,0);
  ...
```

Listing 3.4: An example of #pragma immovable. When values are asynchronously copied to the same variable multiple times, the later operations will have to be labeled immovable in order to ensure correctness.

Synthesis and each one examines or alters the three major file types. A short overview of each phase is presented here and in more detail in the following subsections.

Analysis The analysis phase collects information which will later be used to inform fusion. From each file it collects small pieces of crucial information:

Application File - The fusion regions or sets of functions which are to be fused in the later stages.

Library File - For each function, KFusion collects which kernels are called with which arguments as well as any synchronization information specified by *#pragma sync [in][out]*.

Kernel File - The arguments to each kernel, the inputs and outputs as specified by *kload* and *kstore* as well as any immovable statements as specified by *#pragma immovable*

Synthesis The synthesis phase uses information from the analysis to inform the fusion process.

Application File - Using the synchronization information obtained from the library file, KFusion takes the fusion regions define by *kfuse* and creates sets of functions and arguments to be fused.

Library File - Using the sets provided in the previous steps, KFusion combines the functions present in each set into a singular monolithic function. It also collects all the kernels and arguments and passes these sets down to the next level. It combines the kernels and arguments into one singular set and creates new code for executing the kernel generated in the next step

Kernel File - Given sets of kernels and arguments to fuse, KFusion creates dependency trees of each set and begins fusion them. This reduces down to a singular monolithic kernel called by the library function.

3.4.1 Phase I: Analysis

During analysis, KFusion collects information from the application in terms of which library functions are called and which arguments are used. Primarily, KFusion determines opportunities for fusion across kernel boundaries. This involves the application, library and kernel files. The analysis algorithms can be seen in Algorithms 2 and 3.

Library Files

In the library files, KFusion parses each function and builds a series of abstract syntax trees representing each function. It also collects any relevant synchronization information. In this form, KFusion is able to determine which OpenCL kernels are being called and with which arguments. The statements of each library function is split into four categories:

Pre-statements which occur before any OpenCL calls. These are typically bookkeeping operations which set variables used by OpenCL later.

OpenCL kernel arguments specifies the arguments to be used by the kernel. Specifically, these arguments identify the variables the kernel itself is operating on. These may be different than the function arguments, if they use GPU data structures when the data resides on the target device. These data structures are typically memory objects which reside in GPU memory, and as such data must be explicitly transferred between them.

OpenCL kernel enqueue operations queue and launch kernels with the series of given arguments. These instructions identify which kernels are being launched.

Post-statements statements which occur after all OpenCL calls have occurred. These operations are generally bookkeeping and cleanup operations.

Algorithm 2 Fusion Library Analysis Algorithm

```
1: Read in library file
2: for Each function F in library file do
3:     read in synchronization any info F.sync
4:     if #pragma sync in then
5:         F.isSyncIn = True
6:     end if
7:     if #pragma sync out then
8:         F.isSyncOut = True
9:     end if
10:    read in function arguments F.args
11:    read in function statements store these in F.pre, F.post, F.clargs, F.clKernels
12:    generate F.clInv from kernels and arguments
13: end for
```

It is from this breakdown that KFusion obtains list of kernels and kernel arguments. KFusion does not require specialized annotations as shown in Table 3.4.

Kernel	Arguments
square_kernel	x→gpu_buffer
square_kernel	y→gpu_buffer
add_kernel	c→gpu_buffer, y→gpu_buffer, x→gpu_buffer
sqrt_kernel	c→gpu_buffer

Table 3.3: List of kernels and which arguments they are called with

Kernel Files

When KFusion examines a kernel file, it relies on the manual introduction of the *kload* and *kstore* annotations with the code that identify when values are loaded and stored. An example of how these annotations are applied is shown in Listing 3.3. Here the initial variables are designated as loads and the final operation is designated as a store operation. Using these designations KFusion can create a table of inputs and outputs to each kernel as shown in Table 3.4.

This annotation markup supports the division of statements into three types by KFusion: **load** operations, **store** operations and **core** operations. The core operations are easily defined as operations which are not loads or stores, but more aptly do the actual work. This division clearly identifies the inputs and outputs of a kernel, as well as the operations carried out within it. Later these inputs and output operations will be used to fuse kernels as necessary.

Kernel	Inputs	Outputs
square_kernel	vector x	vector x
add_kernel	vector c	vector x,vector y
sqrt_kernel	vector x	vector x

Table 3.4: List of kernels and which arguments they are called with

Algorithm 3 Fusion Kernel Analysis Algorithm

```
1: read in kernel file
2: for each kernel K in library file do
3:     read in kernel arguments K.args
4:     for each statement in kernel do
5:         assign to K.loads, K.store, K.core
6:     end for
7: end for
```

Application Files

In the application files KFusion uses one annotation to denote library operations that share data and could be fused as shown in Listing 3.1. This is the only annotation used application developer. The previous annotations where done entirely by the library developer.

This marks a region of two or more library function calls which will then be fused together during the synthesis phase. During analysis, KFusion identifies the function calls in the fusion regions and keeps track of their arguments. This collection of data will then be used to create new functions based on the dependencies between the functions. In the end, there is a list of fusion regions, each with a set of functions and function arguments as shown in Table 3.5

Function	Arguments
square	x
square	y
add	c, y, x
sqrt	x

Table 3.5: List of function calls and arguments to be fused later

3.4.2 Phase II: Synthesis

The synthesis process of KFusion begins again with the application files and works through the information collected during the analysis phase.

Application Files

Synthesis of the application files is relatively simple. KFusion takes the list of called functions and iterates through it creating new function calls. The algorithm for accomplishing this can be seen in Algorithm 5. It attempts to greedily fuse all possible functions into one, breaking fusion up at synchronization points.

This process is informed by the synchronization information acquired in the library analysis. In a function requires a synchronized output, fusion is stopped and restarted at the conclusion of the function. Effectively splitting the fusion and providing a breakpoints. If a function requires synchronized input, the same process occurs, but instead the breakpoint is before the synchronized function. The arguments of the

new functions are the collection of arguments for each individual function with any duplicates removed.

Algorithm 4 Application Level Fusion Algorithm

```
1: Given set of function calls to fuse S and set of functions F
2: Define Fusion U = {}
3: Define Fusion Set T = {}
4: for Call c in S do
5:     lookup function f ∈ F such that f.name = c.name
6:     if f.isSyncIn then
7:         T.append(U)
8:         U = {}
9:     end if
10:    U.append(f)
11:    if f.isSyncIn then
12:        T.append(U)
13:        U = {}
14:    end if
15: end for
16: T.append(U)
17: for set of functions to be fused t in T do
18:    output new function call for t
19: end for
```

In this step KFusion is creating a series of sets of functions and arguments which will later be used to create new library functions and kernels. The output of this process is a list of functions and arguments. For the example, this is shown in Table 3.5. This process will also create a new function call and replace the original set of function calls with a single one. For the example a new function call is shown in Figure 3.2.

Another example using the given synchronization primitives is shown in Figure 3.3. This shows how fusion will be broken up based on hard synchronization requirements. This will reduce performance, but ensure correctness. Because the library developer adds the synchronization annotations, KFusion will break up the function calls into fusible groups with no future work on the part of the application developer.

Library Files

Synthesis of the library files is a somewhat more complex process. First for each function call, KFusion duplicates the called function and replaces its parameter names

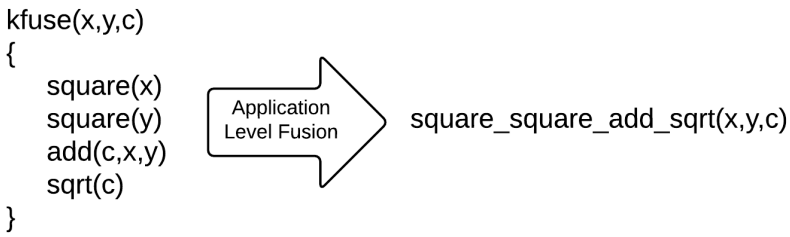

Figure 3.2: Creation of a new function call at the end of the first synthesis stage. The several function calls on the left are replaced by the single one on the right. Duplicate arguments are removed.

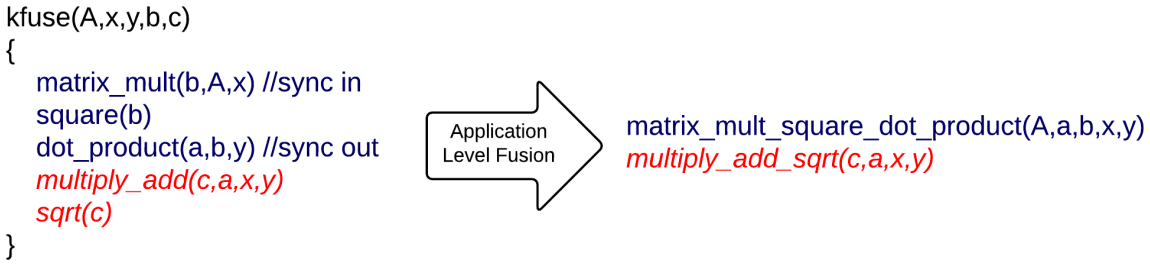

Figure 3.3: Creation of a new function call at the end of the first synthesis stage. This example takes into account the synchronization information given in the application file. In this example a dot product produces a synchronized result and the matrix multiplication requires a synchronized input. When fused, this will create two separate functions. This will ensure correctness.

with generated ones which are guaranteed to be unique for each argument. To achieve this uniqueness KFusion uses an *argument dictionary* which stores all function arguments and maps them to a unique string. This results in a unique function for each call with different parameters. They key reason for doing this is that duplicates can be eliminated. This prevents moving data multiple times or accessing the same value multiple times from global memory. This optimization comes more into play during the kernel fusion, but must be represented here. The argument dictionary can be seen in Table 3.6.

Original	**Mapping**
x	arg_1
y	arg_2
c	arg_3

Table 3.6: The mapping between the arguments passed into each function and the replacement arguments. This ensures each argument is used only once.

KFusion then enters into each function and renames its local variables based on the parameters that affect them. I refer to this as a contamination mechanism. A variable is considered to be contaminated if it is assigned from a parameter or assigned to from another contaminated variable. A contaminated variable contains its regular identifier as well as the identifiers of all the variables which assign to it. This maintains a set of dependencies and approximates data flow as well as ensures each function call has a unique structure. Due to the nature of the C programming language and side effects, this cannot be done perfectly in the general case, but the current KFusion prototype properly alters the code surrounding OpenCL functions with no major problems.

The next step KFusion combines the functions into one. First, the pre and post statements from each function call are combined and any redundant statements are removed. Variables are renamed to reflect their data dependencies ensures that when two of the same functions are called with different arguments, they do not catastrophically destroy each other. While a few extraneous instructions or variables are introduced, this approach safely produces the correct statements with no loss of performance.

Finally, a new *kernel invocation* is created by KFusion. A kernel invocation is the set of functions which define a set of arguments and then execute a given kernel. For each library function, the kernel invocations are extracted and combined to generate

the new invocation. This new invocation contains all the arguments of the child invocations and removes any duplicates or redundancies. The component kernels followed by the new kernel can be seen Table 3.7. Next in the last part of synthesis KFusion will create this kernel.

Algorithm 5 Library Level Fusion Algorithm: The algorithm basically conglomerates each functions code into one, some redundant instructions are removed and a new kernel is generated

```
1: Given set T containing sets of functions to fused
2: for for set of functions S in T do
3:     for function f in S do
4:         replace parameters with mappings in argument dictionary
5:         for For each variable v in f do
6:             if v is assigned from a parameter p, contaminate v with p
7:             if v is assigned to from contaminated variable v2, contaminate v with
   v2
8:         end for
9:     end for
10:    Create a new function fnew
11:    for function f in S do
12:        fnew.paremeters += f.parameters
13:        fnew.preStatements += f.preStatements
14:        fnew.postStatements += f.postStatements
15:        fnew.clArguments += f.clArguments
16:        fnew.clKernels += f.clKernels
17:    end for
18:    Remove duplicate statements in each category
19:    Remove duplicate arguments
20:    Combine kernel invocations into single invocation
21:    Generate new kernel variable
22:    Generate code to load and compile new kernel
23: end for
```

This part of the KFusion process results in a single kernel with all available arguments and only one set of pre instructions and post instructions. The list of kernel executions and arguments, is collected and will be used to generate a new kernel in the next KFusion stage.

The end result of this stage is to create a new library function. It will contain three major components:

1. The combined pre-statements which have duplicates removed and accomplish any necessary initializations.

Kernel	Arguments
square_kernel	x:arg_1
square_kernel	y:arg_2
add_kernel	c:arg_3, y:arg_2, x:arg_1
sqrt_kernel	x:arg_1
square_square_add_sqrt_kernel	**y:arg_2, x:arg_1 c:arg_3**

Table 3.7: The list of kernels with arguments—each forming a kernel invocation—along with the new kernel in bold.

2. The new kernel invocation represented as a series of OpenCL calls setting arguments to a new kernel and the kernel execution itself.

3. the combine post-statements which have duplicates removed and accomplish any necessary cleanup.

Then the set of kernels and arguments is passed another level down to create new kernels as necessary.

Kernel Files

The last stage of the KFusion synthesis is entered with a list of kernels, each called with a unique set of arguments. The analysis phase provided a clearly defined set of inputs and outputs for each kernel. An example of this list can be shown in figure 3.4. The synthesis of the kernel files is similar to that of the library functions in that all kernel parameters are replaced with unique identifiers and local variables are validated to keep track of data flow and to ensure they have unique names. This provides an excellent method of dependency propagation and identifies which outputs are tied to which inputs. Again, similarly to the library functions, a set of unique kernels is created.

Using the additional data-flow information added through dependency propagation, KFusion builds a dependency graph from the bottom up starting at the last kernel executed and moving back to the first. The tree construction algorithm is shown in Algorithm 6. It is a recursive algorithm performing a depth first search in order to match outputs with inputs. Each node is added based on its dependencies. A node is a parent of another node if and only if one of its outputs satisfies one or more of the child's inputs.

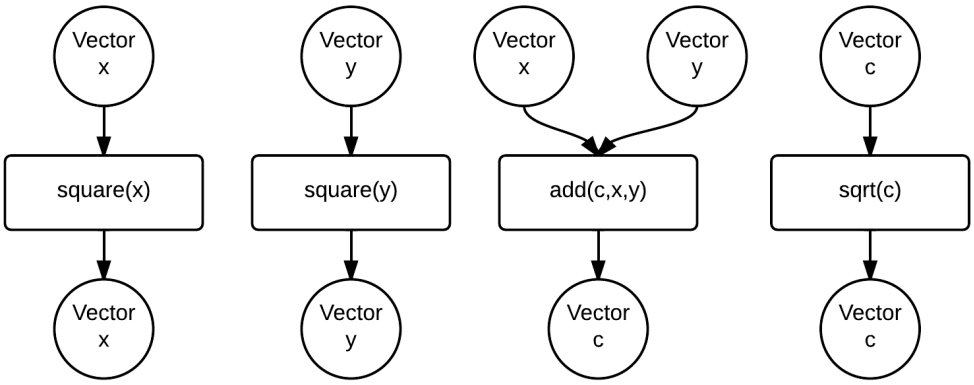

Figure 3.4: An overview of the inputs and outputs to each kernel. These inputs and outputs will later be used to perform kernel fusion accomplishing both loop fusion and deforestation.

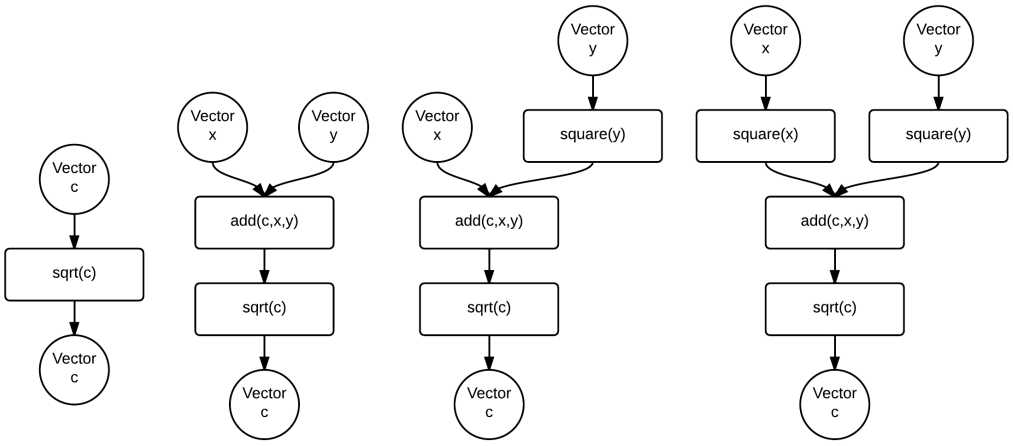

Figure 3.5: An example of constructing the dependency graph given the set of inputs and outputs as shown in Figure 3.4. Inputs are recursively matched with outputs to produce a dependency tree. The inputs are represented by circles and it is relatively easy to see how the data can be pipelined through each operation.

An example of the construction can be seen in Figure 3.5. By adding the kernels in reverse order and always adding them to the newest or topmost possible node in the graph, a dependency graph which accurately represents the correct execution order is created. Interestingly, OpenCL provides an event based synchronization method which allows developers to manually specify dependencies. The KFusion approach does it automatically.

In the event a kernel does not fill any dependency in the graph, it is effectively independent. In this case it is combined with bottom most node. This is accomplished by combining the loads, stores and core operations of each kernel into one. This will add to the number of inputs and outputs of the bottommost node and make it amenable to fusion later as KFusion adds more nodes to the graph.

The dependency graph is then used to correctly accomplish kernel fusion. The algorithm for this process can be seen in Algorithm 7 and an overview of this process is illustrated in Figure 3.6. KFusion scans the input operations of the root kernel and looks for a parent node in the graph with an output that satisfies the given input. If a match is found, the kernel is fused. This starts by replacing any load statements from global variables—which are inputs—with the matching final results of the parent kernel—which are the parent's outputs. This replaces global load operations with copy operations from private memory or local memory. These are much less expensive

Algorithm 6 Kernel Level Fusion Algorithm Part 1: This constructs the dependency graph which will later be used to perform fusion

```
1: Given set of of kernels K_n and dependency tree G is empty
2: for i = n to 0 do
3:     if G is empty then
4:         G_root = Node(K_i)s
5:     else
6:         added = Addparent(G_root, K_i)
7:     end if
8: end for
9: function ADDPARENT(node n, Kernel k)
10:    for Each parent node p in node n do
11:        added = Addparent(G_root, K_i)
12:        if added then
13:            return True
14:        end if
15:    end for
16:    match = False
17:    for output in k.outputs do
18:        for inputs in n.inputs do
19:            if input == output then
20:                match = True
21:            end if
22:        end for
23:    end for
24:    if match == True then
25:        n.parents += k
26:        return True
27:    else
28:        return False
29:    end if
30: end function
```

and avoid many problems which can arise from global memory access. This reduction significantly cuts down on the I/O involved.

Algorithm 7 Kernel Level Fusion Algorithm Part 2: This takes the dependency graph and reduces it to a single node

```
1: Given a dependency graph G and root node R
2: while R.parents ≠ 0 do
3:     for parent p do
4:         if p.outputs match at least one of R.inputs then
5:             for each assignment statement s in R.loads do
6:                 if Right hand side of s is in p.outputs then
7:                     Replace right had size of s with corresponding output
8:                 end if
9:             end for
10:            R.load = p.loads + {p.core} + R.load
11:            Remove p from parents
12:            Add p.parents to parents
13:        else
14:            These graph nodes cannot be compacted, return an error.
15:        end if
16:    end for
17: end while
18: Move all root level declarations to the top of the scope
19: Move all asynchronous operations to below the declaration
20: Output new kernel
```

Then the both the parent kernel's load and core operations are appended to the start of the root kernels load operations. This effectively adds another set of loads as well as dependencies to the root kernel. At this point the parent node's parent are assigned to the root node, then the root node is discarded.

Some bookkeeping is required. The transformer uses scope to mitigate any similarly named local variables from the parent kernel, and a single assignment maybe necessary to move between scopes. KFusion also initiates a cleanup operation to remove any self assignments. Any additional single assignment instructions should be taken care of by the compiler afterwards. This process is repeated until the dependency graph is reduced to a single node.

Once reduced to a single node, KFusion can move statements around in order to improve the execution. Declarations at the kernels root scope are moved to the top of the scope in order to improve readability and ensure variables are declared before they are used. Asynchronous instructions at the base scope are rearranged and moved

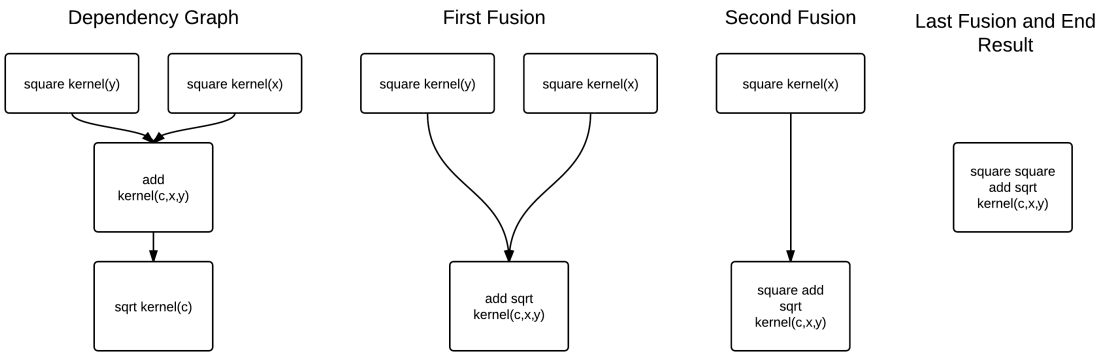

Figure 3.6: An overview of fusing the dependency graph. At each stage KFusion eliminates a parent node of the root node and then reassigns the parent nodes to the root. This collapses the tree creating a fused kernel. While it may initially appear that the inputs and outputs are unchanged, each kernel has to reload the inputs its inputs and store the outputs. When reduced to a mono-kernel, everything is loaded and stored only once.

just after the declarations. These instructions are typically used to move data from global to local memory, executing them early will hide any latency associated with loading values. Moving asynchronous instructions earlier in the execution is a major benefit and an optimization that could not occur with smaller modular kernels. The larger kernels become, the more asynchronous instructions that can schedule early.

If the dependency graph cannot reduce to a single node, the fusion cannot be completed. I refer to this as the *highlander rule*: there can be only one. If there is more than one node in the dependency graph, the current KFusion prototype returns an error. An alternative for future implementations is to return a set of partially fused kernels. Failing hard at this stage is most beneficial as it provides feedback to the application developer attempting to fuse incompatible library functions. The alternative is failing softly which could leave the user with inexplicably inferior performance. Generally, when working with parallelism and concurrency, my experience has been that failing hard is preferable. A tree which cannot be completely compacted should be avoidable using a combination of synchronization annotations at the library level and fusing and automatically combining any independent kernels which cannot be fit into the tree.

3.4.3 End Result

Three artifacts are generated for each successful fusion:

New function call - exists in the application file which leverages the newly created function. This passes the required arguments into the library function and eliminates several function calls and kernel executions.

New function - in the library file a new function which performs the necessary operations to execute a single monolithic kernel. The new function takes care of any and all bookkeeping required by the new underlying kernel. This accomplishes deforestation at a library level as any redundant computation is removed.

New kernel - built from a dependency based fusion. This new kernel accomplishes both deforestation, loop fusion as well as reorders asynchronous instructions. It removes redundant computation and allows for improved data flow. Values stay in cache and local memory longer and throughput is dramatically increased. Data can be loaded asynchronously hiding latency and improving throughput.

In ideal cases this kernel can accomplish the work of its N component kernels in the time it would take to execute a single one.

These new structures are created via a source-to-source transformation method, and the application developer can read the end result and perhaps make further changes and optimizations. At the very least it should allow the developer reason about the transformations and gain further insight about program behavior. This is a desirable feature especially when considering the possibility of daisy chaining further optimization tools. It also ensures portability and most devices will have their own OpenCL compiler invoked at runtime.

Though the KFusion approach introduces trade-offs with respect to information hiding, it allows an application developer to get a high-level overview of the low-level processes a library provides. Leveraging the level of abstraction the library supports, allows the isolation of groups of operations and recognition of data-flow.

3.5 Limitations

KFusion is a proof of concept and has some limitations. The first is that is can only travel down one level of abstraction. This means it is unable to handle functions which call other functions which call kernels. While it will not fail in these instances, it cannot fuse kernels which have two levels of indirection. Currently this is not a major problem, but could be in more complicated systems. Expanding KFusion to handle several levels of abstraction may be possible, but the analysis and fusion would be much more complex. I believe that its current capabilities will handle most libraries and are well within the capabilities expected from a proof of concept.

The second major drawback is that the synchronization is currently too coarse grained and this may limit fusion possibilities. The current synchronization primitives should be expanded upon to allow for the specific declaration of which variables need to be synchronized in and out. Once again this is a reasonable extension that would not prove too difficult to implement, but was not done for this proof of concept.

KFusion only performs a static analysis. A dynamic analysis might be more powerful in determining data-flow and what should be fused. How this would help has yet to be determined, but it is an avenue for exploration.

KFusion also requires there to be a common data thread through a series of operations. It will have a harder time doing a relevant—performance enhance—fusion

on a series of absolutely unrelated operations. This is very much a data-flow tool and supports doing many SPMD operations on singular pieces of data. It cannot support other forms of parallelism and concurrency, though there has been some initial efforts to expand KFusion to allow for the fusion two concurrently executing kernels.

KFusion cannot perform kernel fusion on instructions within loops and is limited to more straightforward transformations. This is because a loop will inherently create a series of dependencies on the previous operation which cannot be passed forward in private memory. This means that while KFusion can effectively accomplish loop fusion by combining kernels, it cannot accomplish fusion of loops within the kernels itself.

KFusion may also run into limitations with the GPUs local memory. Local memory is much faster than global, but limited in size. Each work group only has access to a specific amount of local memory. Combining several kernels which all use an independent space in local memory may cause requested memory to exceed the available amount. This problem could be avoided by allowing KFusion to know more about the hardware. This information could be collected automatically using OpenCL.

KFusion requires the convention of having an application file, a library file and kernel file. Deviating from this format or mixing kernel code in with library code will cause significant problems for KFusion. There is an OpenCL style which stores the kernel files as raw strings embedded within library source code. This is not supported currently and most likely will not be. The two formats should be interchangeable enough anyway and support for embedded kernels will not improve the findings of this work.

Chapter 4

Case Studies

This chapter examines the various case studies which were applied to KFusion: image manipulation, linear algebra and a physics simulation. These provide three key examples covering large areas where KFusion could be useful. Image manipulation provides the best performance case and shows and how to improve operations using KFusion which are driven by data flow. Linear algebra provides another good performance case, but also highlights how correctness has been maintained and how fusion can be applied to wide range of problems and fields. Lastly my physics implementation is a pool game which uses OpenCL to handle all the game-play mechanics and physics. While somewhat simple, this shows how KFusion can operate at the heart of a larger system with many components and still achieve a performance increase.

4.1 Image Manipulation

To evaluate the KFusion prototype, I developed an OpenCL image manipulation library. It is intended work as a standard OpenCL API which leverages a layer of abstraction to separate the low level implementation details from the applications-specific operations. A variety of libraries have been developed which have this model such as ViennaCL [43]. Such libraries are becoming popular, bringing OpenCL performance to various domains, without requiring domain experts to become OpenCL experts.

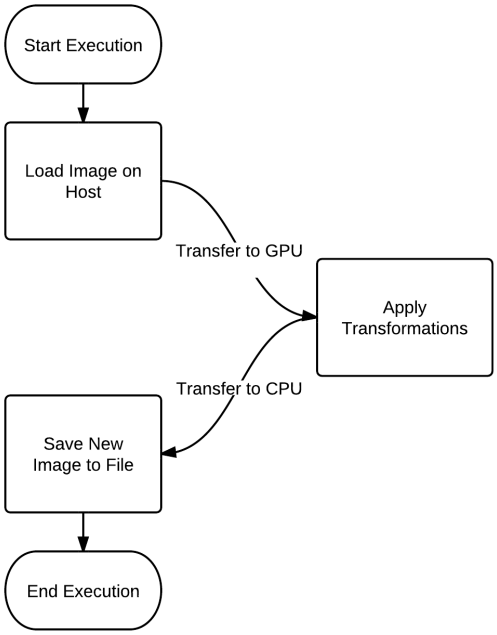

Figure 4.1: This figure shows the general execution model. The application moves data to the GPU, computes a result using the data and then transfers the result back the CPU.

4.1.1 Image Library Design

Operations in the image manipulation library follow the execution model shown in Figure 4.1. An image has to be loaded, moved to the GPU, manipulated, and returned to the host. Here I introduce the data types and operations involved.

Data Types

As the library is written in C, the primary data type is an *Image* struct which stores relevant information such as width/height and pixel values. It needs to maintain a copy of the pixel values on the GPU and transfer the data from host to the device as needed. I implemented a form of lazy copying which ensures data is not transferred any more than necessary. The other major data type is an image convolution matrix which stores the matrix required data for an image convolution on both the CPU and GPU and handles transfers between host and device.

Operations

The representative set of library operations are standard image manipulation operations. While I recognize that improved versions of these operations could be built by domain experts, they suit the purposes of this work when it comes to kernel fusion and provides a accurate representations of the costs of various types of operations. They also provide various forms of dependencies which ultimately determines what can be done do in terms of kernel fusion. I believe they accurately represent a generic set of data intensive parallel operations and that the findings from this case study will further generalize to other domains.

The types of operations can be grouped into three classes based on their dependencies:

Single Pixel Operations are operations carried out for each pixel in the image. These are also fairly essential operations. With single pixel operations the library can convert between color spaces, greyscale and threshold, as well as more complicated operations like adjust color levels within an image. In this case, there is only a single dependency which is an input pixel.

Binary Operations are operations carried out on two images to produce a third. Typically these operations are the standard mathematical ones, add, subtract, multiply and divide, but can be expanded to any operation which has two inputs and produces a single output. In this case the key performance characteristic is that there are now two dependencies as both images need to be ready to create the third one.

Image Convolutions are matrix operations on the pixels. Typically a 3x3 or 5x5 convolution kernel is applied to groups of pixels in the image to generate each pixel in the output image. These operations can be used to blur, sharpen or filter an image. This is how basic edge detection is implemented and image convolutions cover a large array of image manipulation problems. The key performance factor in this area is the need to operate on larger sets of values to produce each output value. This introduces dependencies on at least nine previous values. As such, Image convolutions require synchronization before executing. The alternative requires moving a large amount of operation from one kernel to the next in global memory or repeating a large number of operations. This is generally not feasible.

Operation	Class	Synchronization	Description
ToHSV	Single Pixel	non	Convert to HSV color space
ToRGB	Single Pixel	non	Convert to RGB color space
Colorize	Single Pixel	none	Change all pixels to a particular color
Resize	Single Pixel	non	Re-size Image
Crop	Single Pixel	out	Crop Image
RGBInvert	Single Pixel	non	Invert image in RGB color space
BinOp	Binary Operation	non	Perform binary operation on two images
Convolve	Convolution	in	apply convolution kernel to image
Blur	Convolution	in	apply blur image convolution

Table 4.1: A subset of implemented operations from the image library. These provide a representative set of operations covering those that require very little data, to those requiring a significant amount.

A list of supported operations in the library can be seen in Table 4.2. While there are many single pixel operations, the binary and convolution operations are much more versatile and cover a wide range of functionality.

OpenCL allows for a special image data type, which enables the use of normalized coordinates. This means that some operations which would normally require synchronization—such as resizing an image—can avoid it. In the case of the resize, values are loaded using normalized coordinates and stored using the exact coordinates the final image size. While this works well with images, other data types stored in the image format may require an operation like resize to be synchronized upon completion.

4.1.2 Sample Applications

This case study considers four different application scenarios, which vary according to their inherent dependencies and data flow. In each case, I examine where performance could be improved and reason about the key bottlenecks.

Application 1: Low Data Dependency

The best case scenario in terms of possible gains is one that has few dependencies other than the preceding operations. This means KFusion can really take advantage of fusing kernels. The operations are all single pixel operations and this can be seen in Algorithm 8.

Algorithm 8 LDD Operations

1: Resize the image
2: Convert image to HSV color space
3: Colorize image to a specific color
4: Convert image to RGB color space
5: Invert image

In essence the best reason for combining the kernels in terms of performance is to reduce the memory I/O involved. Each of these operations involves three steps:

1. Load a single pixel
2. Apply a given transformation
3. Store that pixel

When there exists n kernels executing consecutively, they will require n loads and n stores. In terms of this application scenario, that would be 5 loads and stores or ten global memory operations altogether. But with KFusion it becomes possible to combine these operations and amortize these load/store costs. This is shown in Algorithm 9.

Algorithm 9 LDD Fused Operations

1: Load pixel
2: Convert pixel to HSV
3: Colorize pixel
4: Convert to RGB
5: Invert pixel
6: Store pixel in new position with regards to the re-sized image

Application 2: Medium Data Dependency

The average case scenario is one that involves more dependencies, but still allows for reordering to ensure efficient combination of operations. This case involves a convo-

lution as well as a binary operation between two images, as shown in Algorithms 10 and 11.

Algorithm 10 MDD Operations

1: Apply blur convolution to first image
2: Convert first image to HSV
3: Convert second image to HSV
4: Perform binary operation on both images storing the result in the first image
5: Convert first image to RGB

This represents a problem with dependencies and it becomes harder to effectively combine operations in order to improve performance. It is still possible to combine them, but there will not be much of an improvement. Primarily because even if kernels are combined, there will still be a significant amount of global load operations.

Looking at each operation independently it is possible to reason about the costs. Each function requires a similar amount of I/O as in the best case, except for the convolution which requires accessing nine values and writing one, and the binary operation which requires reading two values and writing one.

Algorithm 11 MDD Fused Operations

1: Asynchronously load convolution matrix into fast memory
2: Load second image pixel
3: Convert second image to HSV
4: Apply convolution to second image to produce result pixel
5: Convert result to HSV
6: Apply binary operation
7: Convert to RGB
8: Store result in first image

By combing the functions, KFusion manages to save some memory access and hides some of the latency associated with loading the convolution matrix. This effectively demonstrates the fact that efficient data transfer is a tangled concern.

Application 3: High Dependency

The high dependency case involves two convolutions and very little overlap in computation leading to a smaller amount of fusion. This provides a case where fusion should not make a significant performance increase as there are large number of operations which do not lend themselves to deforestation. The algorithm and operations can be seen in Algorithm 12.

Algorithm 12 HDD Operations

1: Apply convolution to image 1
2: Apply convolution to image 2
3: Apply greyscale to image 1
4: Apply greyscale to image 2
5: Perform binary operation on both images

In this case there is not a large number of operations to chain together. There are two disjoint operations which are brought together to create a single output value. Nonetheless KFusion can take this bare-bones situation and produce a reasonable fusion scenario which may have a slight improvement over the original. The fused result can be seen in Algorithm 18.

Algorithm 13 HDD Fused Operations

1: Asynchronously load convolution matrix for first convolution
2: Asynchronously load convolution matrix for second convolution
3: Apply convolution to image 1
4: Apply greyscale to image 1
5: Apply convolution to image 2
6: Apply greyscale to image 2
7: Perform binary operation on both images
8: Store result

Application 4: High Dependency-Low Fusion

In order to complete the spectrum, the highest dependency case represents a high amount of dependencies leading to very little fusion opportunity. It is a bare-bones case where overlap is minimum, but the kernel needs to access a large number of values. The algorithm and operations can be seen in Algorithm 14. The fused result can be seen in Algorithm 15.

Algorithm 14 HDDLF Operations

1: Apply convolution to image 1
2: Apply convolution to image 2
3: Perform binary operation on both images

Algorithm 15 HDDLF Fused Operations

1: Asynchronously load convolution matrix for first convolution
2: Asynchronously load convolution matrix for second convolution
3: Apply convolution to image 1
4: Apply convolution to image 2
5: Perform binary operation on both images
6: Store result

4.2 Linear Algebra

Linear algebra provides a case which proves that KFusion preserves the correctness of the application. With image manipulation it can prove difficult to judge whether or not fused operations produce the same result as the unfused. With Linear algebra, values can be directly compared and exact results can be confirmed.

4.2.1 Linear Algebra Library Design

Building an OpenCL BLAS library allows for the common operations and datatypes present in linear algebra to be abstracted so the programmer does not have to deal with the internal semantics of every operation. It also allows for the OpenCL details to be abstracted away making it easy for the end programmer to combine operations.

Data Types

In terms of data types, I constructed three major ones: vectors, matrices and sparse matrices. Vectors are set of n values where n is the size of the vector. Matrices are n by m structures common in many applications such as signal processing and computer graphics. Sparse matrices are a special case where the majority of the matrix's values are zero. In this instance only the nonzero values are stored.

Each data type requires two sets of data: one set of values present on the host processor and one on the target device. The data is lazily copied back and forth from the host and device and some additional information is kept on which piece of hardware has the current version of the data.

Operations

In terms of operations, only a few are required in order to develop a library with a wide range of applications. When dealing with image processing it helped to classify each

Operation	Example	Sync	Description
Vector Addition	$c = \alpha * a + \beta * b$	non	Adding vectors a and b where α and β are scalar multipliers
Vector Multiplication	$c = a * b$	non	multiplying vectors a and b
Vector Square Root	$c = sqrt(a)$	non	c is the square root of each element in vector a
Vector Dot Product	$\alpha = sum(a * b)$	out	obtaining the dot product of two vectors a and b
Vector Matrix Multiplication	$b = Ax$	in	multiply a matrix by a vector
Vector Sparse Matrix Multiplication	$b = Ax$	in	multiply a sparse matrix by a vector

Table 4.2: A subset of implemented operations from the linear algebra library. Much like in the image manipulation case study, these provide a representative set of operations covering those that require very little data, to those requiring a significant amount. It is also worth mentioning that while this is a small subset of possible operations these can be used to form larger operations and can be used to build more complex constructs such as solvers.

operation by its dependencies. The same behavior can be seen here, but I classify the operation in a more general case based on synchronization requirements. Operations can require no synchronization or synchronization before or after execution or both. I refer to synchronizing before and after as *in* and *out* respectively.

In this case there are only a few basic operations, but these are the bread and butter of linear algebra and can be used to build larger operations and applications. Most solvers only require these operations. One operation which is most likely missing is a matrix times matrix multiplication. Likewise, an outer product or cross product operation would also be beneficial.

Looking into synchronization, most operations do not require it. They operate on a single value of given data structure. The exceptions are the dot product and the matrix vector multiplication. The dot product requires a global reduction and produces a single value. This means there are no longer have independent operations on parallel values. This will require global synchronization at the end of the dot product kernel. Therefore this is considered a synchronized out operation as synchronization must occur before the next kernel can begin. When looking at the matrix vector multiplication, $Ax = b$, there is the opposite effect. The entire value of the vector x is required before the correct multiplication can be carried out. This also

requires a global synchronization before execution. This creates the synchronized in requirement: fusion must halt at this function as the inputs must be synchronized. Fusion from here on out can be carried out as normal.

4.2.2 Sample Applications

Much like with the image manipulation, I present three examples in increasing complexity and memory access requirements. The first is a simple equation involving a few vectors, the second involves a series of vectors and the third involves a matrix multiplication and a set of vectors.

Application 1: Simple Equation

The first example uses a minimum number of equations to show how fusion is possible within the realm of linear algebra. The algorithm is shown in Algorithm 16. The micro-benchmark solves the Pythagorean theorem for thousands of values in parallel.

Algorithm 16 Operations present in c = sqrt(a^2 + b^2) where a b and c are vectors of length n

1: vector_mult(a2,a,a) - $a2 = a * a$
2: vector_mult(b2,b,b) - $b2 = b * b$
3: vector_add(c2 ,1,a2,1,b2) - $c2 = a2 + b2$
4: vector_sqrt(c,c2) - $c = sqrt(c2)$

While there are only a few operations present here, it is possible to see how they chain together and how fusing the kernels might improve performance. Each operation is accomplished through SPMD parallelism. Each element of the vectors will be operated independently in parallel.

Application 2: Complex Equations: Distance Calculations

The next test uses a larger problem—calculating in the distance between a series of x, y coordinates. The algorithm itself can be seen in Algorithm 17. This has significantly more data, intermediate results and reduces to a set of final outcomes. There are still a small number of dependencies between operations and each element of the result vector can be computed in parallel. Much like the previous, this is also a well suited algorithm for fusion.

Algorithm 17 Operations present in the distance calculations between sets of points $x1$ $y1$ and $x2$ $y2$

```
1: vector_add(dx,1,x1,-1,x2)
2: vector_add(dy,1,y1,-1,y2)
3: vector_mult(dx2,dx,dx)
4: vector_mult(dy2,dy,dy)
5: vector_add(c2,1,dx2,1,dy2)
6: vector_sqrt(c,c2);
```

In this case there are a series of intermediate products and a larger set of operations accessing more memory. This provides a more complicated but well suited problem for fusion.

Application 3: Matrix Equations: Initial Steps of Conjugate Gradient Solver

The final example is the initial few operations of the conjugate gradient algorithm. The other major components of the conjugate gradient algorithm can also be fused, and this provides a representative example of several operations.

Algorithm 18 Initial set of operations for Conjugate Gradient Algorithm

```
1: denseMult(b,A,x);
2: vector_add(r,1,b2,-1,b);
3: vector_add(p,1,r,0,r);
4: vector_mult(r,1,p,1,p);
```

In this case, there is a matrix vector multiplication and the memory I/O for this operation is difficult to mitigate, but KFusion can fuse the following operations resulting in one large mono-kernel. This is effectively the worst case for fusion as the majority of I/O is not mitigated. Likewise because a matrix vector multiplication requires some synchronization, it is always going to be difficult to mitigate the I/O present. Also because most of the memory access associated with the matrix occurs within a loop, it becomes impossible to mitigate the I/O.

4.3 Physics

The physics simulation is a working pool game—slightly simplified in order to reduce development time. It deals primarily with rigid body physics and collisions in a 2D

space. Whereas the previous two case studies involve specialized code for solving specific problems, this case study was designed to show a more general case where a small section of a larger program is sped up using OpenCL and Kernel Fusion.

Data Types

There are two major datatypes: the balls, the table and the cue. Both take the *objects of arrays* approach to performance and development. As opposed to each ball being represented as an object, the container for all the balls is a single object. The description of these data types is as follows:

Balls The Balls object contains the required data to hold an arbitrary number of pool balls. In the case of standard eight ball, this is seven stripes, seven solids, the eight ball and the queue ball. Each ball has a position, velocity, mass, radius and friction coefficient. These are stored in a series of arrays which are mirrored on the target device.

Table The Table contains the dimensions of the pool table as well the location and radius of each pocket. These are also stored in arrays and mirrored on the target device.

Cue The cue is just a set of two values: angle and power. These are not kept in buffers on the target device, but are instead passed in as temporary constant values to each kernel.

Operations

A working game requires a larger set of operations not related to OpenCL and these will not be discussed in detail. Apart from the OpenCL execution of the game's physics, the game must handle and process input as well as render the game to the screen. Because the game is 3D using OpenGL, the drawing functions cannot execute at the same time as the physics functions otherwise the game will lock up.

In terms of OpenCL operations, the game must accomplish these tasks:

1. Hit the cue ball and impart momentum on it
2. move the balls based on their velocities
3. check to see if a ball has landed in a pocket
4. handle collision detection with other balls

Operation	Sync	Description
Impulse(balls, power, angle)	in	Impart velocity to the cue ball based on the current cue angle and power
moveBalls(balls)	out	Move each ball based on velocity
collideBalls(balls)	non	Check for collisions with balls
collidePockets(balls, table)	non	check for collisions with the table pockets
collideWalls(balls, table)	non	check for collisions with the table Walls

Table 4.3: Operations required to implement pool in OpenCL. The collision functions could be easily combined, but this makes it much harder to debug and implement, luckily KFusion can do this for me at compile time

5. handle collision detection with sides of table
6. handle collision detection with pockets

These can be converted into a set of operations as shown here in Table 4.3. Much like with linear algebra, it's worth considering synchronization. Some operations, such as moving the balls require the result to be synchronized before further operations take place. Likewise the impulse operations requires a synchronized input in order to properly start the balls in motion.

It is worth noting that while the collision functions could be combined into one, there are identifiable software engineering benefits to keeping them separate. Primarily, I was concerned with testing each one in order to ensure they operated correctly. This allowed for me to ensure the accuracy of the simulation. Keeping them separate also allows for isolated development and prevents cascading changes. Finally it allows for the ordering of the operations to change depending on requirements and the results of testing.

4.3.1 Application: OpenCL Pool

Unlike the other case studies which have several applications, OpenCL pool is much more focused and as such only has the one. Nonetheless this provides evidence that KFusion is capable of improving the computational heart of a smaller application. This subsection will detail the composition of the functions and other considerations. The game itself can be seen in action at `http://youtu.be/idEr2-gtGTo`.

```
kfuse(g)
{
  collideBalls(&g,timestep);
  collidePockets(&g);
  collideWalls(&g);
  moveBalls(&g,timestep);
}
```

Listing 4.1: The KFusion annotations required to initiate fusion for the physics OpenCL Pool Example

When actually developing the application it is important to order the operations correctly so they are amenable to fusion. The correct ordering can be seen in algorithm 19. This is because *moveBalls* requires synchronization after execution and the collision functions will work best if there is a synchronization just before all of them. When applying fusion, it makes sense to ignore impulse as it is only executed once. Fusing the contents of the while loop provides the best benefit. The resulting fusion annotations can be seen in Listing 4.1

The simulation itself was designed to operate at a very small time-step of 0.1ms. This was primarily done in the interest of accuracy, but also as a good judge of performance. Also as a game-play feature, the player is allowed to see .2 second into the future when judging which shot to take. This makes accomplishing a detailed simulation in a short period of time important.

OpenCL pool leverages OpenGL along with the GLUT toolkit to handle graphics and keyboard inputs and outputs. The OpenCL code handles the games mechanics and is passed into GLUT in the form of a callback function. This will be executed whenever the graphics and input systems are idle. The fact that OpenCL is integrated in a larger application makes this case show how KFusion can improve OpenCL portion of an application which may have other functionality.

Algorithm 19 Initial set of operations for conjugate gradient algorithm

```
1: impulse(balls,clue)
2: while balls are moving do
3:     collideBalls(balls)
4:     collidePockets(balls, table)
5:     collideWalls(balls, table)
6:     moveBalls(balls)
7: end while
```

Chapter 5

Evaluation, Analysis and Comparisons

Previous work has highlighted the need for principled approaches to modularity and information hiding, even in highly optimized code [14, 39, 15]. In a case where an application uses a library, such as in the case studies given in this work, ideally the developer does not need to look any further than the API for the library. Fusion and deforestation applied to the kernels in the library—in application-specific ways—can provide opportunities for significant performance gains. This section evaluates the Kfusion OpenCL-to-OpenCL source to source transformer both qualitatively in terms use of use and software engineering benefits and quantitatively in terms of performance.

5.1 Qualitative Evaluation: Ease of Use and Software Engineering Benefits

The following qualitative assessment focuses on the benefits and costs of the Kfusion approach to loop fusion and deforestation from the perspective of the application programmer. I first consider the benefits in terms of sheer lines of code the application programmer must deal with (Section 5.1) and the annotations they must deal with (Section 5.1.1). This assessment is followed by an investigation of the impact of the KFusion approach on core software engineering principles and a consideration of the tradeoffs encountered and how these issues could be mitigated (Section 5.1.2).

Case Study	Library	Kernel	Fused Library	Fused Kernel	% Increase
Image Manip.	815	322	1034	759	34.5%
Linear Algebra	422	159	626	393	75.3%
Physics	593	232	624	377	26.9%

Table 5.1: Lines of code for fused and libraries and kernels.

Lines of Code

The Kfusion approach achieves the application specific performance benefits of looping and deforestation but eliminates the need to write a substantial amount of code. The benefits of Kfusion not only reduces the lines of code an application developer writes but also reduces overhead in terms of testing and application evolution.

Specifically, the application developer is saved from writing new kernels which in turn require new functions and new functionality within existing functions. Table 5.1 demonstrates the magnitude of the lines of code automatically generated by the Kfusion process with up to a 75% increase in the case studies described in this paper.

In addition to this extra code inside the kernels and libraries, Kfusion automatically generates the configuration code required for each new kernel including: 1) code to compile the kernel at run time, 2) code to assign arguments and execute the kernel and 3) code to transfer data to and from the GPU.

While it can be argued that a manual implementation would require less lines of code, even if half of the lines of code were eliminated, an increase of 12% to 37% increase the kernel and library code, not including the configuration code that would also have to be written.

Because the automatic code generation process of Kfusion does not reorder the lines of code but just regroups the code into monolithic kernels, the functionality and behaviour of the application does not change. Even issues of synchronization of shared data structures are managed by the Kfusion process. It is these characteristics of KFusion's process that result in a code base that requires less testing overhead than that of a manual implementation.

The automatic code generation of the Kfusion process allows the application developer to never have to navigate the generated code. This eases evolution at the application level, allowing the developer to make changes to their application and regenerate the fused code as opposed to evolving both the application and the manually written kernel and library code.

```
kfuse(image)
{
    resize(image, image->width*1.1,
  image->height*1.1);
    toHSV(image);
    colorize(image,color);
    toRGB(image);
    RGBinvert(image);
}
```

Listing 5.1: The use of kfuse in the first application scenario, LDD (best case).

5.1.1 Annotations

The Kfusion approach does require some manual code writing in the form of annotations at the level of the application and the level of the kernel. In terms of the application developer's perspective, I envision the developer only having to deal with the application level annotations while the kernel annotations would be written by the library expert.

Examples of the application level annotations required in the case studies are shown in Listings 5.1 through 5.4. The *kfuse* mechanism demonstrated here, wraps a set of library function calls that perform computation on a group of data. This approach does require the application level programmer to have some knowledge of the underlying library as only library functions that contain OpenCL kernels can be fused. This forces the application programmer to be aware of which library functions contain kernel functions.

The annotations required at the kernel level require knowledge of the underlying library and associated kernels but as described above as such a library specialist would work at this level. For example, it is necessary to mark *load* and *store* operations as well a certain asynchronous instructions which should not be optimized by Kfusion. Even these points are well defined and distinguishable by a library developer who would identify a load as any assignment from a global value and a store as any assignment to a global value. An example of the use of the *kload* and *kstore* annotations at the kernel level are shown in Listing 5.5.

Table 5.2 Provides an overview of the number of annotations that were required for three of the case studies described in this paper.

```
kfuse(image1,image2)
{
    convolve(image1,blur);
    toHSV(image1);
    toHSV(image2);
    binOp(image1,image2,AVG);
    toRGB(image);
}
```

Listing 5.2: The use of kfuse in the second application scenario, MDD.

```
kfuse(image1,image2)
{
    convolve(image1,blur);
    convolve(image2,sharpen);
    greyscale(image1);
    greyscale(image2);
    binOp(image1,image2,SUB);
}
```

Listing 5.3: The use of kfuse in the third application scenario, HDD

```
kfuse(image1,image2)
{
    convolve(image1,blur);
    convolve(image2,sharpen);
    binOp(image1,image2,SUB);
}
```

Listing 5.4: The use of kfuse in the fourth application scenario, HDDLF (worst case).

Case Study	Kernel and Library Annotations	Application Annotations
Image Manipulation	20	4
Linear Algebra	12	3
Physics	8	1

Table 5.2: Lines of code for fused and libraries and kernels.

```
__kernel void RGBtoHSV(__read_only image2d_t
      input, __write_only image2d_t output){
   kload(input){
      const sampler_t smp = ...;
      float2 coord = (float2)
         ((float) get_global_id (0),
         (float) get_global_id (1));
      coord . x /= get_global_size(0);
      coord . y /= get_global_size(1);
         float4 val =
            read_imagef(input,smp,coord);
   }

   val = toHSV(val)

   kstore(output){
      int2 coord2 = (int2)
         (get_global_id(0),get_global_id(1));
      write_imagef(output,coord2,val);
   }
}

__kernel void colorize(__read_only image2d_t
      input, __write_only image2d_t output,
     const float color){
   kload(input){
      const sampler_t smp = ...;
      float2 coord = (float2)
         ((float) get_global_id (0),
         (float) get_global_id (1));
      coord . x /= get_global_size(0);
      coord . y /= get_global_size(1);
         float4 val =
            read_imagef(input,smp,coord);
   }

  val.x = color;

   kstore(output){
      int2 coord2 = (int2)
         (get_global_id(0),get_global_id(1));
      write_imagef(output,coord2,val);
   }
}
```

Listing 5.5: The use of kload and kstore to annotate the kernel cod in two kernels which will be fused. It becomes apparent just how much redundant operations are removed though fusion

5.1.2 Maintaining Software Engineering Principles

In terms of software engineering principles, the Kfusion approach allows an application developer to gain performance while: 1) retaining modularity, 2) maintaining the stable library API, 3) supporting evolution and change and 4) concealing underlying complexity in comparison to that of manual fusion.

In terms of modularity, all of the original library and kernel modules are kept in tact with just the addition of the annotations described above. It is only at compile time that the extra code is generated and a single monolithic kernel is produced for execution purposes. This monolithic code is never intended to be navigated or viewed by either the application or library developer. In a manual approach, the developer would have to both write and navigate a monolithic implementation in order to achieve these performance improvements.

With the compile time generation of the Kfusion approach, the library user sees no change to the API with the temporary addition of functions and kernels added by compilation. This approach provides all the benefits of a fixed API, including compatibility with other applications using the library.

Evolution and maintenance is also supported with the Kfusion approach at both the application and library level. If changes or updates are made to either the application or the library and kernel code, recompilation will allow the Kfusion process to automatically apply to the new code base. Because the library and kernel modularity is retained after the Kfusion process from both the application and library developer's perspectives, all the benefits of evolving a modular code base are maintained.

While the application developer is required to have some knowledge of which library functions contain kernels, they are shielded from the complexity of synchronization and data structure details. That is, an application developer does not have to worry about managing synchronization of shared data structures and they do not need to be knowledgeable about the types of operations and data structures that map to the GPU to apply performance enhancing techniques.

As described above, an application developer must identify high-level opportunities for fusion using the *Kfuse* annotations and therefore requiring some knowledge of the underlying library, subsequently breaking rules of strict information hiding. Successful fusion and performance boosts can only be obtained if there is a thread connecting each successively executed library operation. While this can be fairly obvious in some domains, such as image manipulation, it may be much more difficult

in others. In order to mitigate this issue, I believe that future development of the Kfusion process could include the integration of application level visualization tools that would highlight opportunities and hints for function combinations that could leverage fusion. This kind of tools support could be populated with the help of a library developer that would identify through a rule-based system, library functions that group well together.

5.2 Quantitative Evaluation: Performance

For all three applications there is a fairly significant performance increase in most cases and fusion never produces a result which performs worse than the original unfused kernels. In this section, the performance results for each case study are examined. In each case there are several types of applications which increase in dependencies and reduce the available fusion. Each case study was executed on the Nvidia Tesla and Intel i7 in order to gain a better understanding of how fusion effects both types of hardware.

In each subsection, I present the GPU results and CPU results in terms of speedup as well as provide a comparison between the CPU performance and GPU performance.

5.2.1 Experimental Design

Each case study had each test case executed 1000 times and the execution times of the kernels were collected and averaged to produce the final results. The large number of iterations were used in order to obtain a reliable result independent of various factor which could influence performance. OpenCL kernels need to be executed a few times to *warm up* and execute at their fastest. I measured the actual execution time of each kernel as oppose to including the transfer time moving data to and from the GPU. While the transfer time may be large in some cases, a pipelined approach combined with asynchronous communication can be used to mitigate this potential area of performance loss. Also while transfer time should be considered when deciding to use a GPU as a target device for ones computation, this is not the focus of this work.

I tested the kernels on two platforms. The first was a NVidia Tesla C2075 Co-processor [30]. It is a high end NVidia Co-processor which possesses 448 cores, 6GB of memory and has a peak double precision floating point performance of 515 Gflops.

It is the most recent Tesla processor from NVidia and in many ways is equivalent to a high end GPU. The largest difference is it has more memory than is typically available in standard GPUs.

The second was an Intel i7 2600K processor [18]. It is a four core processor with 8 hardware threads. It has a maximum clock rate of 3.4 GHz and posses an 8 MB L2/L3 cache. While the Tesla operates as an extremely high end GPU with a much larger memory, the Intel i7 is a standard CPU processor. This provides contrast and provides support for the generalization of these results. The following subsections give the performance results of each case study. When executing on the Intel CPU, timing anomalies would occur and which produced the occasional result which could be several orders of magnitude slower than the average. These results were removed from the final results as they are clearly

The test system ran Ubuntu 11.10 (oneiric) with kernel 3.0.0-24-generic. It had 8 GB of memory. To compile and execute OpenCL code I used the nvidia compiler driver (NVCC) as part of the cuda 4.2, V0.2.1221 toolkit. NVCC will use the available underlying compilers, which on the test machine where g++ version 4.6.1 and gcc version 4.6.1. The two compiler switches uses where: -lstdc++ and -lOpenCL.

5.2.2 Image Manipulation

Image manipulation provides the best performance case. A lot of data is involved in each image and the operations can be combined relatively easily. This ensures data is kept in fast memory for as long as possible and load and store operations to global memory are kept to a minimum.

GPU Results

With regard to GPU performance, in the best case (Application 1, Low Data Dependency or LDD) Kfusion achieve approximately a four times speedup and in the worst case (Application 4, High Data Dependency Low Fusion or HDDLF) there is a much smaller, but still moderately significant performance increase. The results in terms of relative speedup can be seen in Figure 5.1.

The best cases have minimal dependencies and achieve a significant performance increase over the unfused implementation—up to four times faster. With medium and high dependency cases there is much less of an improvement, one and a half times faster. These cases have much more data movement which cannot be removed through

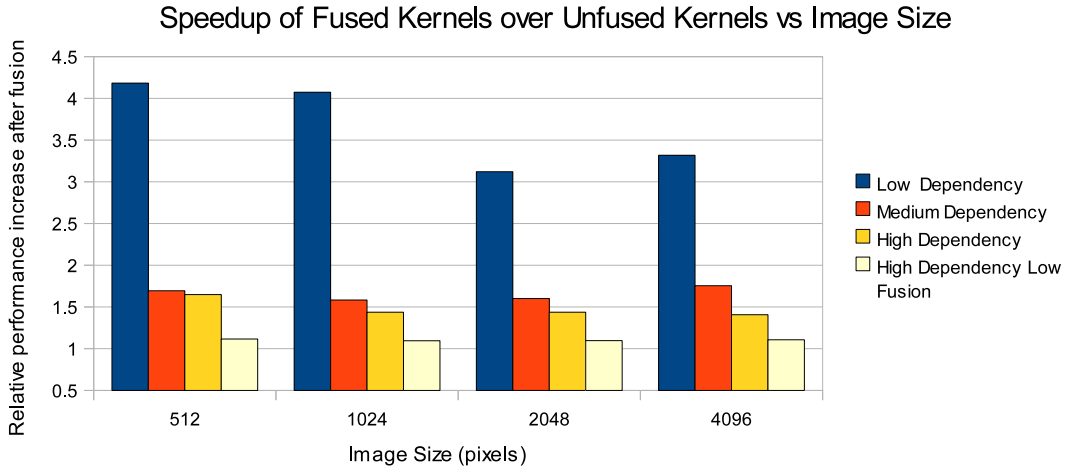

Figure 5.1: Image Manipulation GPU Results: The relative speedups of fusing in each application scenario for various image sizes. In all cases there is a performance increase and Kfusion achieve the largest when there is minimal dependencies

fusion and deforestation. In the worst case, where there is a high amount of memory I/O and only a single fusion operation there is much less of a performance increase. This shows that, even working with a high-level construct like *kfuse*, opportunities for loop fusion and deforestation can produce a corresponding performance increase.

To show the effectiveness of Kfusion, results of automatic fusion are compared to ones obtained by hand fusing kernels. This can be seen in Figure 5.3. In general the transformer performs very similarly to hand fused kernels. In the low dependency case Kfusion actually outperforms manually fused kernels and as the image size increases, the transformer scales better than and outperforms hand fused kernel code. I do not have results for the low dependency case with an image size 2048 by 2048 pixels. This is due to hand fused code having erratic performance with this image size. It is several orders of magnitude slower than it should be when given a 2048 by 2048 image. In this way Kfusion produces more reliable code.

CPU Results

The CPU results with regards to image manipulation are very similar to the GPU results and bring support to the generalizable nature of this work. The results are shown in Figure 5.3. In the best case there is still a large performance increase and as dependencies increase and less I/O can be mitigated, performance decreases. The

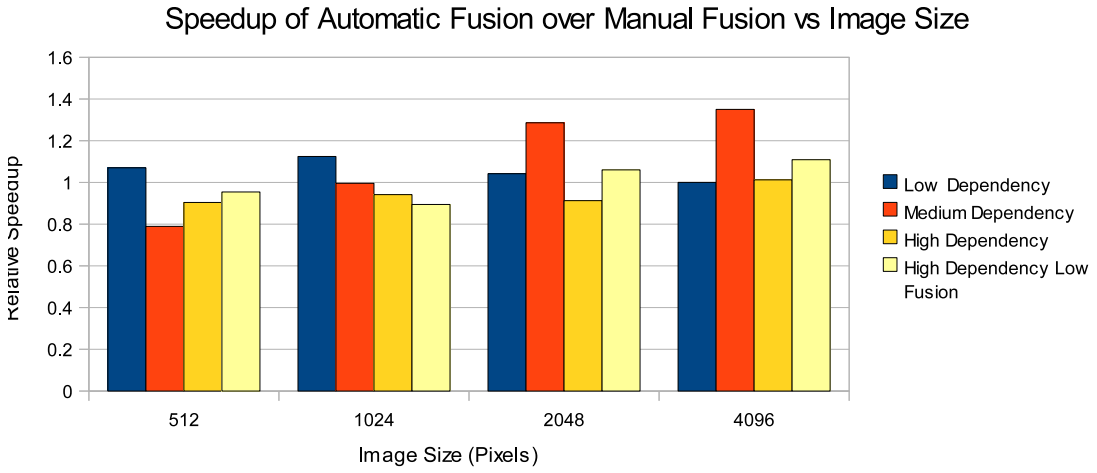

Figure 5.2: Image Manipulation GPU Results: The relative speedup of automatic fusion vs manual fusion. We see that automated fusion performs comparatively to manual fusing kernels and occasionally exceed the performance of manual fusion. This shows the viability of the fusion approach.

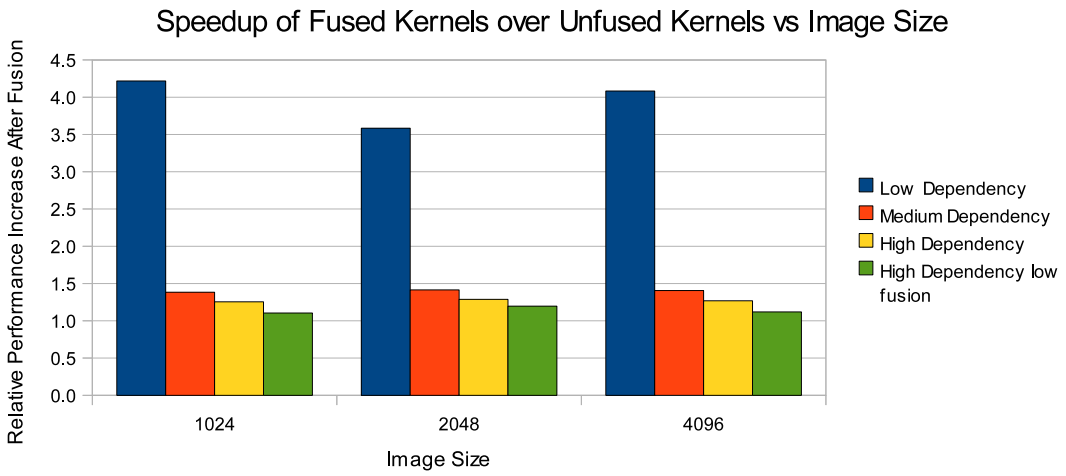

Figure 5.3: Image Manipulation CPU Results: There are very similar performance increases as with the GPU. Fusion produces the largest increases in the event of minimal dependencies. As dependencies increase performance gains are reduced.

results comparing automatic fusion to manual fusion can be seen in Figure 5.4. In this case study, the automatic fusion performs no worse than manual fusion on the CPU. In some cases the automatic fusion is superior, but this is not a regular occurrence.

Comparison of CPU and GPU results

Table 5.3 compares the CPU results to the gpu results. Moving image manipulation to the GPU can improve performance by 2 orders of magnitude. This is because of specialized hardware which can take advantage of memory locality as well as a high level of available parallelism. In all cases, fusion increases performance.

5.2.3 Linear Algebra

Examining the linear algebra case study results, there are similar performance gains, though often not as drastic. This examines several low dependency cases and as dependencies are increased, there is a reduction in performance.

GPU Results

When operating on the NVidia Telsa there are large gains as I/O can be minimized between kernels. This is especially true when dealing primarily with vectors. Fusion

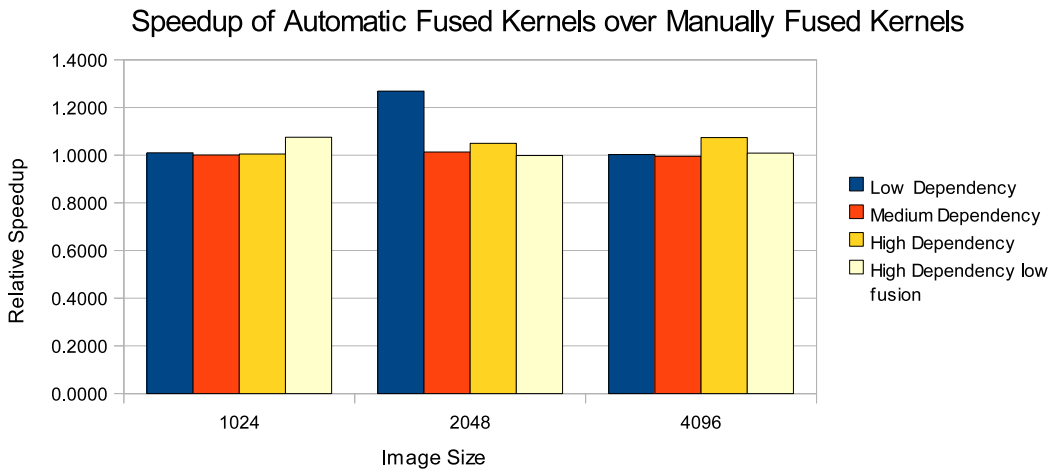

Figure 5.4: Image Manipulation CPU Results: The automatic fusion performance in most cases with the exact same performance of the manually fused kernels. In some cases the automatic fusion produces superior results.

	Execution Time (ms)			
Case	Unfused CPU	Fused CPU	Unfused GPU	Fused GPU
Low Dependency	3155	773	99	29
Medium Dependency	3236	2301	114	65
High Dependency	4873	3842	143	101
High Dependency Low Fusion	4241	3789	236	213

Table 5.3: Image Manipulation Results: A comparison between the CPU execution vs the fused GPU execution time with a 4096x4096 image. In the best case performance increases by over two orders of magnitude.

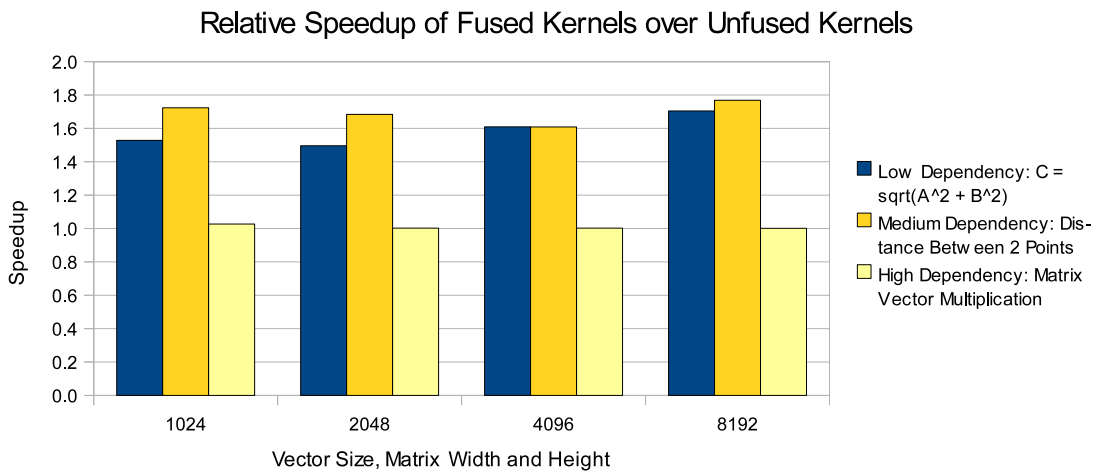

Figure 5.5: Linear Algebra GPU Results: The relative speedups of fusing in each application scenario using differing data sizes. In the cases when dealing primarily with vectors there is a significant performance increase. In the event a Matrix is used, the performance increase is overshadowed by the execution time of Matrix operations as most I/O cannot be mitigated. This is perhaps an are for future research.

and deforestation remove intermediate results and many operations. The exception is the third application example. In this case the performance increase for fusion is very small. Much like with low fusion case of the image manipulation case study, the matrix vector multiplication provides creates an unfortunate situation where there is a large number of I/O operations that cannot be mitigated through fusion.

Matrix-vector multiplication requires many values to be loaded from memory that will only be used once and none of the matrix values can be maintained in memory through the execution of the following kernels. While it is possible to keep the vector data in memory and accomplish fusion and deforestation this way, the matrix I/O dwarfs the performance gains that are possible from fusion. As such, there is almost not measurable increase in performance. As the matrix becomes larger, it continues to dominate performance more and more as shown in the tests. At 1024, there is still a small performance increase and at 8192, this is more of less nonexistent. Nonetheless, this example shows the applicability to fusion to standard linear algebra operations which covers a wide range of problems in computing.

Next, Figure 5.6 shows a roofline model [52]. The roofline model is based on the concept that peak performance is limited by the minimum of the available bandwidth and maximum floating point operations. As such, performance is dominated by the

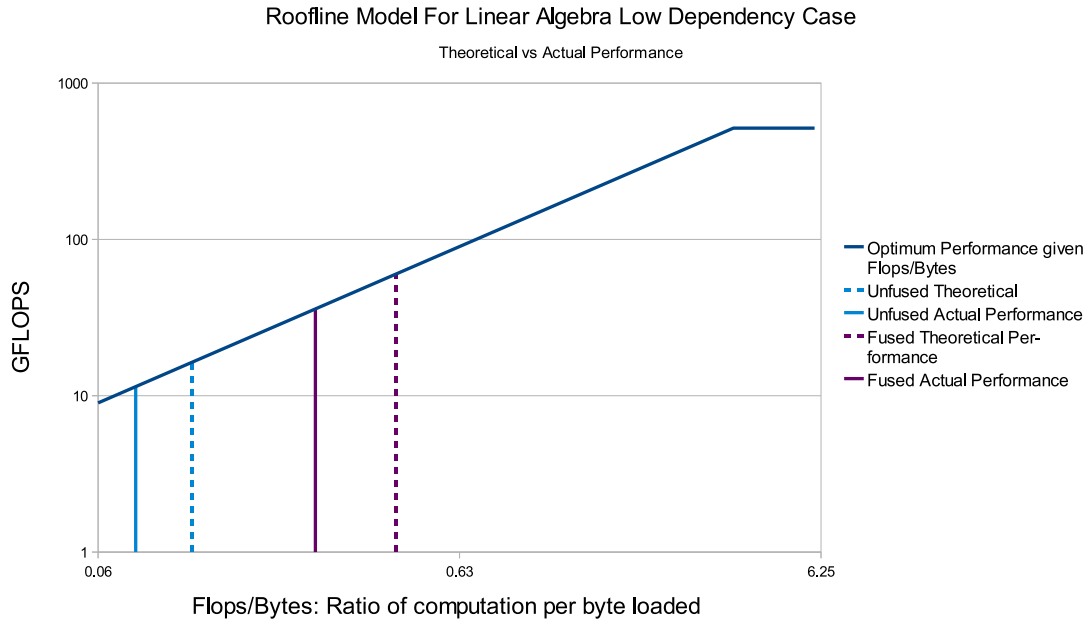

Figure 5.6: Linear Algebra GPU Results: Roofline model for low dependency case. This compares the peak theoretical performance for both the unfused and fused implementations with the actual performance.

amount of data which needs to be loaded in order to complete the floating point computations. The model gives the maximum possible performance based on the ratio of floating point operations to bytes loaded.

The figure compares the theoretical peak performance of the unfused and fused implementations in terms of GFLOPS. While the unfused and fused implementations do not operate at peak performance, the actual performance boost is similar to the theoretical improvement. This also shows the KFusion improves efficiency over the standard implementation. Using KFusion breaks the theoretical rooftop present over the individual modules. This is extremely beneficial and shows the power of KFusion.

Implementation	Theoretical Performance	Actual Performance
Unfused Kernel (GFlops)	16.36	11.44
Fused Kernel (GFlops)	60	35.93
Performance Increase	3.66×	3.14×

Table 5.4: Linear Algebra Results Roofline Table. This table displays the theoretical and actual performance for the unfused and fused implementations.

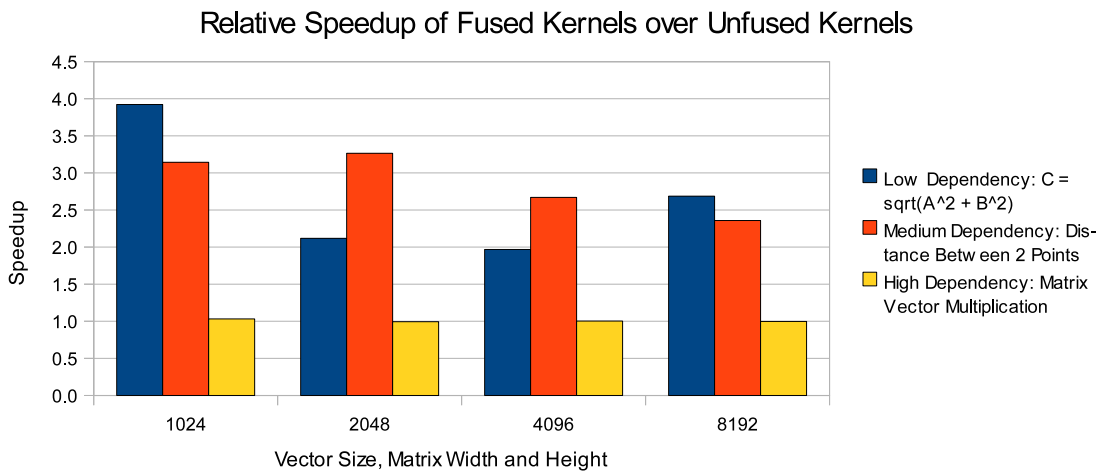

Figure 5.7: Linear Algebra CPU Results: The relative speedups of fusing in each application scenario using differing data sizes. With the CPU results there is a much larger relative performance increase

Table 5.4 provides the performance of both the fused and unfused kernels in terms of GFlops. This provides both the actual performance and theoretical performance as well as the comparable speedups.

CPU Results

The CPU results as shown in Figure 5.5. In the low dependency case, there is a close to four times performance increase. This is really interesting and somewhat unexpected. Loop fusion and deforestation are perhaps even more useful techniques when applied to standard CPU architecture. Data can stay in registers and cache longer and this has the potential to greatly improve performance even with very few cores.

In both the low and medium dependency cases, there is a large performance increase due to the loop fusion and deforestation removing a large number of load and store operations. In the high dependency case, there is a small almost negligible performance increase. Matrix-vector multiplication contains a significant amount of load operations which cannot be mitigated.

	Execution Time (ms)			
Case	Unfused CPU	Fused CPU	Unfused GPU	Fused GPU
Low Dependency	.12	.06	.06	.04
Medium Dependency	.13	.05	.06	.04
High Dependency	102.35	102.-8	3.53	3.52

Table 5.5: Linear Algebra Results: A comparison between the CPU execution vs the fused GPU execution time with a 4096 element vectors and 4096x4096 matrices. In the best case, moving computation to the GPU improves performance by two orders of magnitude.

Comparison of CPU and GPU results

Table 5.5 compares the CPU results to the GPU results for linear algebra. While fusion is beneficial, moving the vector based computation to the GPU does not provide as large an increase in performance. Without considering transfer times, moving matrix-vector multiplication to the GPU on the other hand produces a speedup of 2 orders of magnitude. These transfer times —which may be mitigated through pipeline operation on many images— can greatly reduce the performance boost. More information on this can be found in the work by Lee et al[27].

5.2.4 Physics

The physics case study involves very few kernels which execute extremely quickly. The execution times for each individual kernel are extremely small mostly because they don't access a lot of data. Nonetheless, fusion provides a reasonable performance increase. This gain comes from two major sources: First, the fused kernel executes fewer instruction and accesses more data. Second, there are fewer kernels to enqueue and execute. This saves time which would be wasted between kernels. Both the CPU and GPU results can be seen in Figure 5.8. In both cases there is a significant relative performance increase, with the CPU getting a much better relative gain in performance.

This case study shows that OpenCL core of a program can be fused without interfering with the other components. One side benefit to fusing kernels in an application which is also using the same device to display graphics, is there is a reduced chance of conflicting graphics and compute kernels being scheduled in a way which causes deadlock. Likewise a single kernel is much easier to schedule and execute.

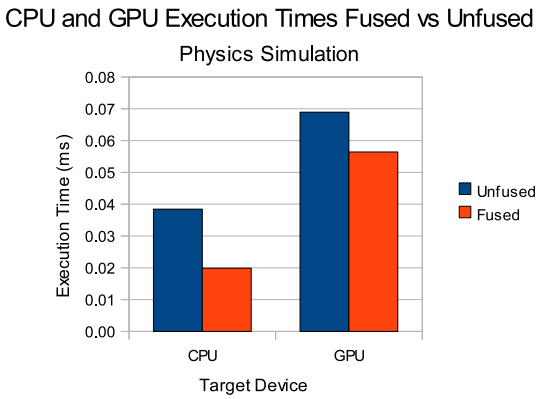

Figure 5.8: The speedup of fusing the physics operations in OpenCL pool. There is a significant speedup for both GPUs and CPUs.

Chapter 6

Conclusions

This work explores the age old tension between modularity and performance in the context of modern parallel programming with a focus on general purpose GPU computing. To address modularity and performance concerns, the KFusion OpenCL source to source transformer was built. The KFusion prototype allows developers to maintain kernel boundaries for the purposes of conceptual representation, but leverages application-specific information to eliminate them for the purposes of performance. I demonstrate that with semi-automated tool support, it becomes possible to effectively address this problem through a novel application of loop fusion and deforestation to GPU programming.

KFusion was tested on three OpenCL libraries developed as case studies which accomplish: Image manipulation, linear algebra and physical simulations. Each library highlights a specific aspect of what KFusion can do well. The image manipulation case study provides a case where performance can be greatly increased, while linear algebra provides a verifiable correctness case and the physics simulation shows how a small section of a larger application can be improved with KFusion.

In terms of performance, KFusion provides significantly improved performance over unfused kernels and performs similarly to hand fused kernels — sometimes outperforming them. In my case it studies never harms performance – which is extremely important. This should apply to most applications, but there may be pathological cases. In the event the fused operations have minimal dependencies and a large amount of I/O can be mitigated, KFusion can provide a large performance increase. The potential performance increase is limited by the number of dependencies between operations and how much I/O can be mitigated. KFusion functions best when large amounts of I/O can be mitigated.

In terms of correctness, KFusion produces correct results. Synchronization annotations prevent incorrect fusions from occurring and kernel level annotations ensure the correct inputs and outputs are matched.

In terms of qualitative metrics. Compared to manually fused libraries, using KFusion reduces code size significantly. This translates roughly into the programming effort that would be required to build each fused kernel manually. In the worst case, combining kernels increased code size by approximately 75%. The amount of annotations required to generate the fused kernels is very minimal. Typically KFusion requires two annotations per kernel and at most one annotation per function.

In terms of maintaining the principles software engineering such as abstraction and information hiding, one can combine kernels without knowing their internals or implementation. This maintains the libraries API, but it is important to know each kernels inputs and outputs. This is an improvement over manual fusion which would require breaking the API and altering a libraries internals. KFusion extends a given API, using it as building blocks for new functions which are created on demand at compile time. Because KFusion extends an API, using components from it KFusion maintains the basic software engineering principles while improving performance. KFusion also produces readable source code.

KFusion has limitations. It can only handle one level of abstraction and expects on OpenCL kernel launched in each library function. I feel this is reasonable for a proof of concept implementation. It cannot do kernel level transformation within control structures such as loops. The optimizations it can accomplish within a kernel is in fact somewhat limited as it primarily replaced global memory operations with much faster private memory operations and moves asynchronous communication around. I believe that this limitation does not harm the work, as KFusion was designed to accomplish inter-modular optimizations, which it does.

In conclusion, the KFusion source-to-source compiler generates monolithic kernels from low level modular components while maintaining the original semantics. This improves performance on-par with manually fused kernels while maintaining software engineering goals.

Future work includes determining the specific ways in which my approach is complementary to other code generators and optimizing compilers, further improving the state of the art. For example, combining this work with an optimizing source-to-source compiler that operates at an intra-kernel level would prove beneficial. This should allow for the greatest possible automated optimization and further allow for

modular development and monolithic performance. As mentioned previously a process involving (1) optimization, (2) fusion and (3) re-optimization is most likely ideal. In this way KFusion could either incorporate an optimizing compiler or an optimizing compiler could expand to incorporate KFusion. The end result of this work could result in an extension to an existing OpenCL compiler.

Another avenue is to explore further optimizations that exist between kernels, and the ways in which these may be realized at a high-level. The next step along these lines is to start fusing kernels to allow for concurrent execution of multiple kernels. I effectively combined kernels vertically to allow for shared data between kernels. Another avenue is to fuse kernels so they both execute at the same time even if they do not share data. This should solve another inherently tangled concern on GPUs: under-utilization of available processors. A GPU can have thousands of cores. If we have a set of operations which do not leverage all available cores, they could be combined horizontally in order to cross the available work space. This most likely is not applicable to images as even a small two dimensional objects can have thousands of data points, but it could work well with linear algebra systems where operations like dot products reduce to a singe value. This also opens the idea of recursively fusing kernels both vertically and horizontally. Most likely this could only go so far, as OpenCL has no global synchronization mechanisms within kernels.

Associated with these avenues of future work would be the development of tool support. Augmenting existing IDEs to help developers navigate and trace from high to low-level, and most effectively establish the right balance of optimizations in their code. Tis may compromise information hiding, but most likely not isolation of concerns. Further studies will also be necessary to establish if the compromise in information hiding may in fact pose a barrier to adoption.

KFusion uses a semi-automated approach to fuse kernels to provide performance and maintain software engineering benefits. KFusion's design leaves the onus on the user of a given library to explicitly identify fusion opportunities based on and the data involved in computation. Transformations come from application-specific information, and result in transformations at the level of application, library and kernel files.

The alternative would be using a fully automated approach that did not require annotations on behalf of the application and library developers. The synthesis process could be converted to a bottom up process, starting at the kernels and moving up to a single function call. This could allow for a fully automated approach. Though this

resolves the compromise with information hiding and would be less onerous for the developer, it also introduces further trade-offs that must be considered.

First, there still may be a need for the library developer to create a set of hints to constrain the transformation space, but this would require minimal input from the application developer. Second, an application developer is effectively using a constantly changing and optimizing library, which may have other consequences with respect to the need (or not) for optimization. When the developer has to specify fusion regions using the *kfuse(param)* notation, they are experimenting consciously. This requires an understanding of the underlying process, but also ensures the ability to more clearly trace the impact of optimizations. They can reason about the changes and replicate them with consistency.

Currently KFusion gives the responsibility to the application developer to determine what is fused and what is not. Ideally documentation and tool support can provide the knowledge required to initiate beneficial fusions. It may be possible in future work to consider an approach supporting both automated and informed fusion operations through the use of library level hints and user level annotations.

Bibliography

[1] Westgrid GPU Computation, 2012.

[2] AMD. Amd radeon hd 6990 graphics. http://www.amd.com/us/products/desktop /graphics/amd-radeon-hd-6000/hd-6990/Pages/ amd-radeon-hd-6990-overview.aspx#3. date accessed: July 2011.

[3] AMD. OpenCL Memory Model. http://www.amd.com/us /products/technologies /stream-technology/ opencl/pages/opencl-intro.aspx. date accessed: July 2011.

[4] AMD. *ATI Stream Computing OpenCL*. 2010.

[5] David F. Bacon, Susan L. Graham, Oliver, and J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, 1994.

[6] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, chapter 14, pages 244–263. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[7] Michael G. Burke and Ron K. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.*, 39(4):139–154, April 2004.

[8] 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.

[9] 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.

[10] Loc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation: a deforestation case-study, 1999.

[11] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5:46–55, January 1998.

[12] Ole-Johan Dahl and Kristen Nygaard. SIMULA. In *Encyclopedia of Computer Science*, pages 1576–1578. John Wiley and Sons Ltd., Chichester, UK, 2003.

[13] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[14] Edsger W. Dijkstra. The structure of the "THE" multiprogramming system. In *Proceedings of the first ACM symposium on Operating System Principles*, SOSP '67, pages 10.1–10.6, New York, NY, USA, 1967. ACM.

[15] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall series in automatic computation. Prentice-Hall, 1976.

[16] Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. In *the Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing, ICASSP '98*, 1997.

[17] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[18] Intel. Intel core i7-2600k processor (8m cache, 3.40 ghz). http://ark.intel.com/products/52214/Intel-Core-i7- 2600K-Processor-%288M-Cache-3_40-GHz%29, 2011. date accessed: July 2011.

[19] Intel. *The Intel Xeon Phi Coprocessor: Parallel Processing, Unparalleled Discovery.* 2012. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.

[20] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[21] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, UK, 1994. Springer-Verlag.

[22] 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.

[23] Khronos. OpenCL. http://www.khronos.org/opencl/, 2011.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[25] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, February 1974.

[26] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[27] 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. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[28] Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 91–100, New York, NY, USA, 2009. ACM.

[29] Bil Lewis and Daniel J. Berg. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

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

[31] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[32] Barbara Liskov. In *The Second ACM SIGPLAN Conference on History of Programming Languages April 20 – 23, 1993, Cambridge, United States*, New York.

[33] Wen-Lin Liu, Chung-Horng Lung, and Samuel Ajila. Impact of aspect-oriented programming on software performance: A case study of leader/followers and half-sync/half-async architectures. In *COMPSAC*, pages 662–667, 2011.

[34] Tim Mattson and Kurt Keutzer. Our pattern language (opl). In *Workshop on Parallel Programming Patterns (ParaPLOP)*, March 2009.

[35] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[36] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[37] Cedric Nugteren and Henk Corporaal. Introducing 'bones': a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[38] NVIDIA. *NVIDIA CUDA Programming Guide 4.2*. 2012.

[39] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[40] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, February 2004.

[41] Andreas Resios. GPU Performance prediction using parametrized models. Master's thesis, Utrecht University, The Netherlands, 2011.

[42] Arch D. Robison and Ralph E. Johnson. Three layer cake for shared-memory programming. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 5:1–5:8, New York, NY, USA, 2010. ACM.

[43] Karl Rupp. ViennaCL. http://viennacl.sourceforge.net/. date accessed: July 2011.

[44] Karl Rupp, Josef Weinbub, and Florian Rudolf. Automatic performance optimization in viennacl for gpus. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[45] Edwin H.-M. Sha, Edwin H.-M. Sha, and Nelson L. Passos. Efficient polynomial-time nested loop fusion with full parallelism, 1999.

[46] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[47] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.

[48] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM.

[49] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[50] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248,

Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[51] David Walker and David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.

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

[53] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.