NTNU

Norwegian University of
Science and Technology

# Auto-tunable GPU BLAS

**Jarle Erdal Steinsland**

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

# Problem description

This project focuses on developing techniques for auto-tunable BLAS. In particular, it looks at such BLAS not only for CUDA, but also OpenCL. One or more such BLAS routine(s) will be developed and tested in the framework.

Assignment given: 17. January 2011
Supervisor: Dr. Anne Cathrine Elster, IDI

# Abstract

OpenCL is fast becoming the preferred framework used to make programs for heterogeneous platforms consisting of at least one CPU and one or more accelerators. The GPU being readily available in almost all computers, it is the most common accelerator in use.

Good libraries are important to reduce development time and to make particular development environments, such as OpenCL, useful for the masses. All OpenCL programs can execute on any device that have support for it, however to achieve optimal performance, a OpenCL program must be optimized for a specific device.

Auto-tuning is a strategy to automatically generate and find a good performing program for a specific device, without requiring the user to perform optimizations manually.

BLAS contains routines that are useful for many algorithms suited for GPUs, and is a good candidate for a library that can prove useful for many OpenCL programmers.

We have chosen, in this thesis, to implement the matrix multiplication routine from BLAS as it is important for the performance of many higher-level linear algebra algorithms to have a fast implementation of matrix multiplication. The exact operation we have implemented is $C = alpha * A * B + beta * C$, were A, B and C are MxK, KxN and MxN matrices respectively.

In this thesis, we implement an auto-tuning framework that generates source code for OpenCL kernels and find the best one for the device it is being executed on.

We compare our version with ViennaCL, a OpenCL BLAS library, and the vendor provided BLAS libraries provided by AMD and NVIDIA. Our

version provides approximately 85% of the performance of the vendor specific library provided by NVIDIA, in general, and gives a speedup over the native library provided by AMD. This speedup is usually between 1.5 and 2. On both platforms our version outperforms ViennaCL by a large margin.

# Acknowledgements

I wish to thank Dr. Anne Cathrine Elster for being my supervisor. I would also like to thank NVIDIA for donating GPUs to the HPC-Lab at IDI, NTNU through Dr. Elster's membership in their Professor Partnership program. Finally I would like to thank my friends and family for their support through my work on this thesis.

Trondheim, June 2011

Jarle Erdal Steinsland

# List of Algorithms

# List of Figures

# List of Tables

# Contents

# Chapter 1

# Introduction

OpenCL is becoming the preferred framework used to create programs that execute on heterogeneous platforms. A heterogeneous platform consists of at least one CPU and one accelerator. GPUs are the most common and easily available accelerator.

For a framework such as OpenCL to be useful to a large number of people, it is necessary to have libraries available to perform commonly used functions while achieving good performance across all devices used. All programs written in OpenCL can run on all devices that support OpenCL, but to get good performance, they must be optimized for the specific devices. This makes creating such libraries a difficult and time consuming task.

Auto-tuning is a strategy that have been used with success in the past on CPUs, and several studies suggest that it would work well on GPUs as well.

BLAS is one such library that provide functionality that is heavily used in many algorithms that are well suited for the GPU. Having a good performing BLAS library implemented in OpenCL would therefore be an important addition to anyone using OpenCL to program GPUs.

General matrix multiplication is one of the routines in BLAS that is most important to achieve good performance in other higher-level linear algebra algorithms. It if for this reason that we have chosen matrix multiplication as the routine to be implemented in an auto-tuning framework, in this thesis. More specifically, we chose to implement the operation $C = alpha * A * B +$

$beta * C$, where A is a MxK matrix, B is a KxN matrix and C is a MxN matrix. Alpha and beta are scalars.

## 1.1 Outline

The rest of the thesis is organized as follows:

Chapter 2 contains background information related to this thesis and a brief overview of related work.

Chapter 3 describes our implementation of an auto-tuning framework. It also goes into specific detail on the code generator generating the matrix multiplication kernels and how the framework searches for the best solution.

Chapter 4 contains benchmarks and results from executing the kernel chosen by the auto-tuning framework on different GPUs.

Chapter 5 concludes the thesis and presents future work.

Appendix A presents the source code selected by the auto-tuning framework for the different GPUs.

# Chapter 2

# Background

This chapter presents background information and previous work. More specifically, we start off by describing what is meant by an auto-tuning system and what components it consists of. Then we present BLAS and introduce OpenCL and the abstractions and models used within that framework, before we describe the graphics processing unit (GPU) architecture of the two major vendors, AMD and NVIDIA, and the characteristics of these that affect performance. Finally, we present some of the previous work that is related to this thesis.

## 2.1 Auto-tuning System

An auto-tuning system, or a system for automated empirical optimization of software (AEOS) as it is called in [1], consists of three main parts[1]:

1. A method of adapting software to differing environments
2. Robust, context-sensitive timers
3. Appropriate search heuristic

Auto-tuning as a strategy for finding good ways to perform an operation for a particular architecture relies on being able to perform the particular

operation in many different ways. There are mainly two different ways of providing this.

The first and simplest way is to have a fixed code or algorithm with parameters in it. By changing these parameters different versions of the code is generated. The other way is to have a highly parameterized code generator that can generate a vast number of ways to perform the same operation.

A timer is needed to find the best code version. A particular code version is chosen for a particular architecture by executing the different code versions on that architecture, measuring their execution time and choosing the best one. As the load of the machine can vary at any given time, the timers must be robust enough to not be affected and produce correct timings irrespective of machine load.

Dependent on how how many possible code variations there are, searching through all of them can be time consuming. The job of the search heuristic is to search through the different code versions and find the best one without taking to much time. The more possible code versions, the more important it is for the search heuristic to be able to prune the search tree quickly.

## 2.2 Basic Linear Algebra Subprograms

Basic Linear Algebra Subprograms (BLAS) is an application programming interface (API) for linear algebra libraries[2]. The functionality provided by BLAS is separated into three levels. BLAS level 1, contains functions to perform scalar, vector and vector-vector operations[3]. The level 2 BLAS provides functions for matrix-vector operations and level 3 supplies functions for matrix-matrix operations.

The reference implementation for BLAS is available as a free download at [4]. In addition to the reference implementation, several vendor specific implementations are available. See [3] for a list of some of them.

## 2.3 Open Computing Language

The Open Computing Language (OpenCL) is a framework for developing programs that perform general-purpose computations on all available processors and accelerators in a heterogeneous platform[5]. It was originally developed by Apple and was later made into a proposal in cooperation with AMD, IBM, Intel, and Nvidia before it was submitted to the Khronos Group. On November 18, 2008, the specification for OpenCL 1.0 was released. Version 1.1 of the OpenCL specification was released on June 14, 2020 and added features for added flexibility for the programmer and increased performance.

The OpenCL framework consists of the OpenCL C Language Specification and the OpenCL Runtime API Specification[6]. The OpenCL Specification[7] use a collection of four models to describe the underlying ideas of OpenCL. These models are the platform model, the memory model, the execution model and the programming model.

### 2.3.1 The Platform Model

The platform model is comprised of a host that have one or more devices connected to it. The host executes the application and issues commands to the connected devices to perform computations on them.

A device is further split into one or more compute units and the compute units are split into processing elements, see Figure 2.1. It is the processing elements that perform the actual computations that the host have requested.

### 2.3.2 The Memory Model

The memory available to work-items executing a kernel[1] on a OpenCL device is divided into four separate memory spaces. These are global, constant, local and private memory. In addition there are OpenCL images, which reside in global memory, but have some special properties that regular global memory buffers do not. See Figure 2.2 for a conceptual diagram of the OpenCL device

[1] An OpenCL kernel is a function that is executed on a OpenCL device.

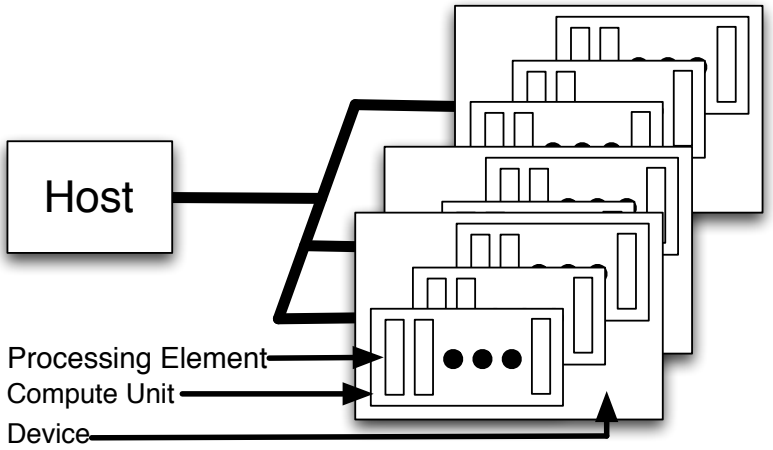


Figure 2.1: Conceptual Diagram of OpenCL Platform Model[7].

architecture and how the different memory spaces are related to the platform model.

It is the host that allocates global memory on the device by creating memory objects. Once created, the host can issue commands to the device to perform various operations on these memory objects. For instance, it can issue a command to copy data from the host memory to the memory object in global memory on the device.

### Global Memory

All global memory is available to all work-items on a device and both reading and writing is supported. If it is supported by the device, reads and writes to global memory are cached.

OpenCL images are a special type of global memory buffers. Unlike regular global memory buffers, which store their elements sequentially, images store their elements in a format that is hidden from the user. Therefore, reads and writes to images must be done by using the built-in functions supplied by OpenCL for this purpose.

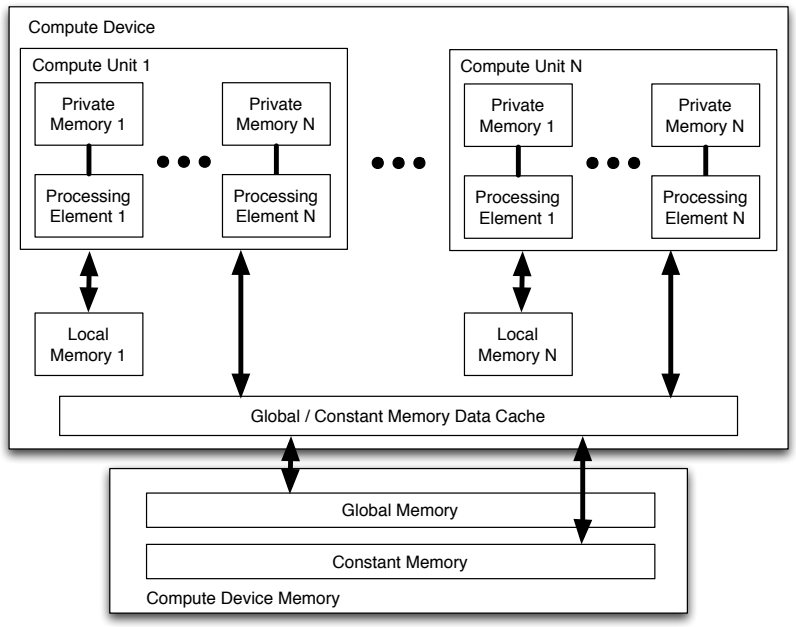


Figure 2.2: Conceptual Diagram of OpenCL Architecture and How the Memory Model Relates to it[7].

### Constant Memory

Constant memory is located in a region of global memory and can as a result be read by all work-items executing a kernel on a device. It have the special property that is does not change during the execution of a kernel. Thus, reads from constant memory can be optimized for quick access on devices that support it. For instance through a special constant memory cache.

As constant memory can only be read by the work-items on a device, it is both allocated and initialized by the host.

### Local Memory

Local memory can either be a separate dedicated memory region on the device or it can be located in a region of global memory. Local memory is only accessible to work-items within a work-group. Thus, local memory is a good candidate for data that is shared among work-items in a work-group.

This is especially true on devices where local memory resides in a separate dedicated memory region, as this memory region is often implemented by memory chips with lower access times than those used for global memory[8].

#### Private Memory

Private memory is a region of memory that is accessible by a single work-item. It is the default memory space used for variables in a kernel when no other memory space is specified and is usually used for temporary storage of values that are specific only to one work-item.

#### Memory Consistency

According to the OpenCL Specification[7], the OpenCL memory model have a relaxed consistency. This means that there is no guarantee that the memory state is consistent for all work-items at all times. As a result, if two work-items reads the same location in memory at the same time, the value read by the first work-item can be different from the value read by the second.

The memory consistency guarantees given by OpenCL are:

- Global and local memory is consistent for all work-items within a work-group at a barrier
- Memory load and store consistency within a work-item

### 2.3.3 The Execution Model

The OpenCL execution model divides the execution of an OpenCL program in two. There is the part of the program that executes on the host and there is the part that executes on a device. The part that executes on the host is no different from any other program and follows the execution model that is used on the host architecture.

To submit a kernel for execution on a device the host must create an OpenCL context by using functions provided by the OpenCL Runtime API. An OpenCL context is an object that contains the resources needed to execute a kernel. These resources being the devices the host intends to execute

kernels on, the kernels to be execute, the program object that contains the source code and executable for the kernels and the memory buffers needed by the kernels.

When the host have created the context, it must create an OpenCL command-queue that is used to schedule kernels for execution and other commands to a device.

An index space, called an NDRange, is defined when the host submits a kernel for execution on a device. For each point in this index space, one instance of the kernel is executed. Each such instance is called a work-item and is uniquely identifiable by its index. The index space is further divided into a more coarser index space. Each point in this coarser index space is called a work-group and each work-group is a collection of work-items from the larger index space. The work-groups are uniquely identifiable by their index in the coarse-grained index space and each work-item within a work-group is also uniquely identifiable within the work-group. See Figure 2.3.

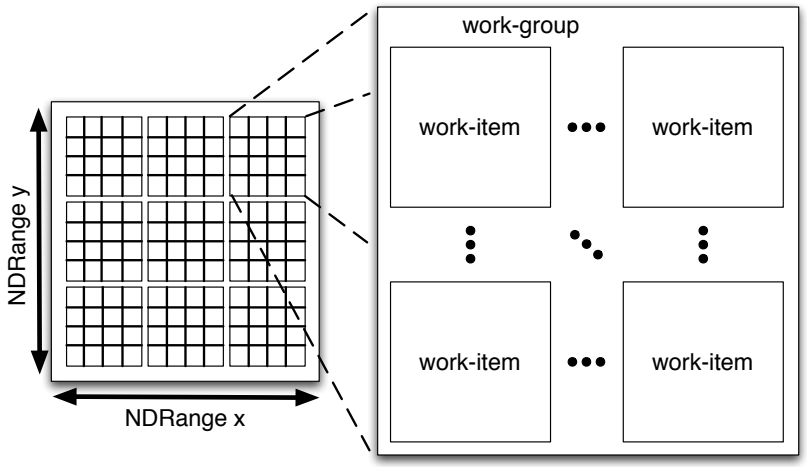


Figure 2.3: An NDRange index space divided into work-groups and work-items[7].

### 2.3.4 The Programming Model

There are two types of programming models that are supported by OpenCL. These are, the data parallel and task parallel programming models. In addition any hybrid of these are also supported.

In the data parallel programming model, a series of computations is defined and is then performed on several several elements in memory. How the mapping of memory elements to work-items is done is defined by the index space that is generated when a kernel is scheduled for execution.

The task parallel programing model is the equivalent of the data parallel programming model with an index space with one work-group and one work-item. Only one instance of a kernel is executed on a device and parallelism is achieved by using vector datatypes and by scheduling several tasks at the same time.

### 2.3.5 The OpenCL C Programming Language

OpenCL kernels are written in the OpenCL C programming language. The OpenCL C programming language is based on the ISO/IEC 9899:1999 C language specification[7], also known as C99.

The features that where removed from C99 are[9]:

- Standard headers
- Function pointers
- Recursion
- Variable length arrays
- Bit fields

The features that where added are[9]:

- Vector types
- Synchronization

- Address space qualifiers
- Many built in functions (e.g. work-item manipulation, math)

## 2.4 AMD Architecture

The first graphics processing units (GPUs) of AMDs Evergreen architecture was released in the fall of 2009 and the rest followed with the last being released in February 2010[10].

A GPU consists of several compute units (also called SIMD Engines) and each compute unit comprise 16 stream cores, which consists of five processing elements, depending of the GPU model[2][11]. See Figure 2.4 for a diagram of the GPU architecture and Figure 2.5 for a diagram of a stream core. At each cycle, every stream core within a compute unit perform the same instruction in a lock-step fashion. These instructions are issued to the processing elements by one very long instruction word (VLIW).

All of the processing elements can perform single-precision floating point operations and the fifth processing element in a stream core can also execute transcendental operations[3]. To perform double-precision operations, two or four of the non-transcendental operations capable processing elements are combined.

Every compute unit have 32 kB of local, on-chip memory called local data share (LDS) and a 8 kB L1 cache. L2 cache is shared by several compute units. The local data share is divided into 32 memory banks[4], that are four bytes wide and 256 bytes deep. One memory operation can be performed for each bank each cycle, but if more than one operation maps to the same memory bank, a bank conflict occurs and the operations are serialized.

A compute unit also have 256 kB of registers available. The register space comprise 16384 general purpose registers, where one register contains four 32-bit values.

[2]The low-end GPUs have only four.
[3]E.g. sqrt, log, sin, cos
[4]The lower-end GPUs only have 16 memory banks.

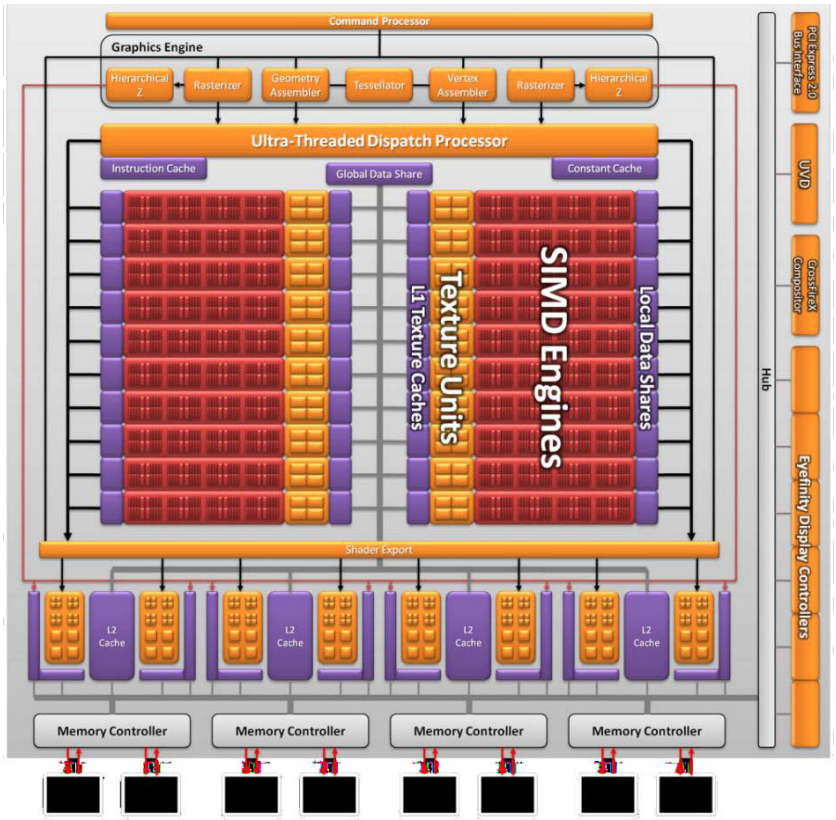


Figure 2.4: Diagram of the AMD Evergreen GPU Architecture[12].

The first GPUs of AMDs Northern Island architecture were released in October 2010[14]. The remainder of the GPUs from this series were released toward the end of the year and in the spring of 2011, with the last one being released in April.

The low and mid-end devices kept the 5-way VLIW of the Evergreen architecture, but the high-end devices switched to a 4-way VLIW instead as studies showed that the usage rate of the processing elements were only 3.4 in games[13]. In the 4-way VLIW design, the processing element capable of computing transcendental functions were removed and its functionality distributed to the remaining four processing elements. The memory cache and local data share stayed the same as for the Evergreen architecture. For a diagram over the Northern Island architecture, see Figure 2.6. Figure 2.7

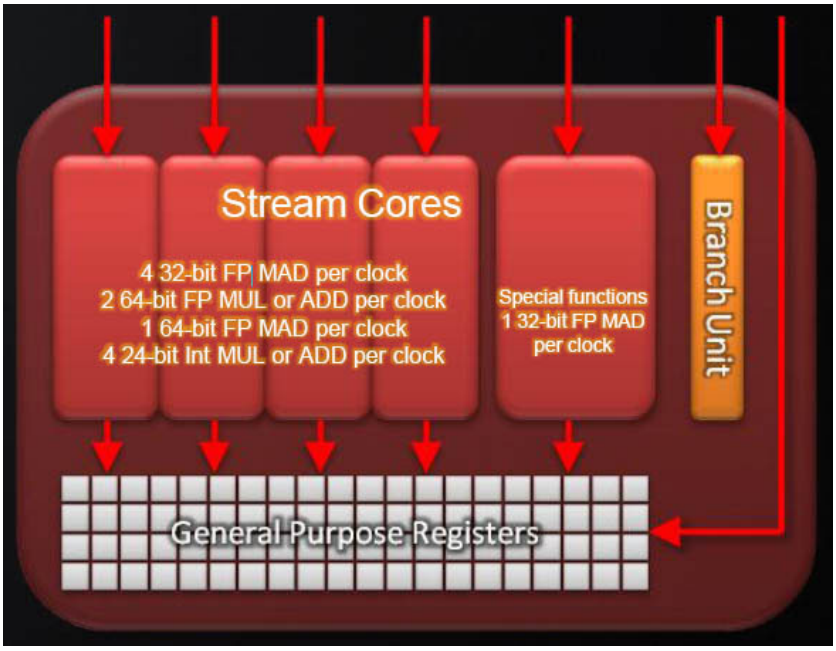


Figure 2.5: Diagram of a Evergreen Stream Core[13].

shows a diagram of a Northern Island stream core.

### 2.4.1 OpenCL on AMD GPUs

When work-items are executed on a GPU they are divided into groups of 64 work-items called wavefronts[5] that runs in lockstep on a compute unit. Every work-group is divided into an integer number of wavefronts and to achieve optimal performance, the number of work-items within a work-group should be divisible with the wavefront size.

As a kernel is being executed, a work-group is assigned to a single compute unit and a work-item runs on a stream-core. Four work-items from the wavefront being executed are pipelined on one stream core to hide memory latencies. At each cycle, 16 of the work-items in a wavefront execute one instruction. When a wavefront is looked at as a whole, this give the appearance that one instruction is executed every four cycles. If the execution paths of

[5]A wavefront is 32 work-items on low-end GPUs.

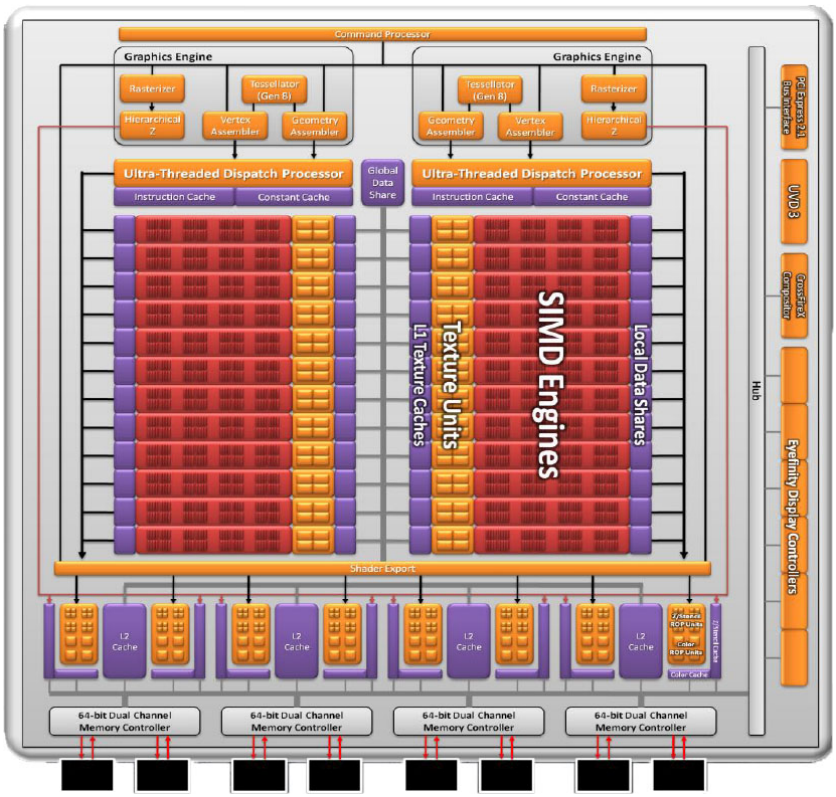


Figure 2.6: Diagram of the AMD Northern Island GPU Architecture[13].

work-items within a wavefront diverges, their execution are serialized.

Private memory usage in kernels is mapped to the general purpose registers as much as possible. If not enough registers are available, the compiler will generate spill code and remaining private memory need is placed in global memory.

It is the local data share that is used for OpenCL local memory. Ensuring that all memory banks are used and that there are no bank conflicts, are the most important aspects with regards to achieving optimal performance.

To achieve good performance with global memory, it is important to ensure that memory reads are coalesced. Coalesced reads occur when consecutive work-items in a wavefront read consecutive elements from memory. Coalesced writes writes can also give better write performance, however, this is of less importance as the difference is quite small[11].

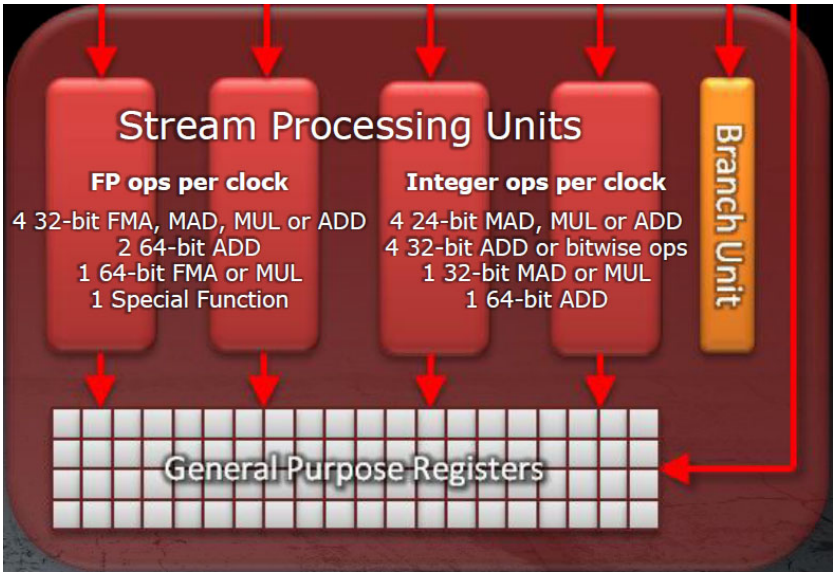


Figure 2.7: Diagram of a Northern Island Stream Core[12].

## 2.5 NVIDIA Architecture

The first GPUs of NVIDIAs Fermi architecture were released in April of 2010, with the rest following throughout the spring and fall[15].

A GPU consists of several graphics processing clusters (GPC). Each graphics processing cluster is made up of four streaming multiprocessors, which comprise 32 cores[16]. See Figure 2.8. In addition to the 32 cores, each streaming multiprocessor also contains four special function units that can perform transcendental functions and 16 load load and store units enabling a streaming multiprocessor to calculate 16 source and destination memory addresses per clock cycle. Each core contains a fully pipelined integer arithmetic logic unit and a floating point unit. See Figure 2.9.

All streaming multiprocessors have 32768 32-bit registers and 64kB of on-chip memory[17]. The on-chip memory can be configured as 16kB of shared memory and 48kB of L1 cache or as 48kB of shared memory and 16kB of L1 cache. Additionally, all the streaming multiprocessors share 768kB of L2 cache.

NVIDIA released the first GPU of an updated Fermi architecture in

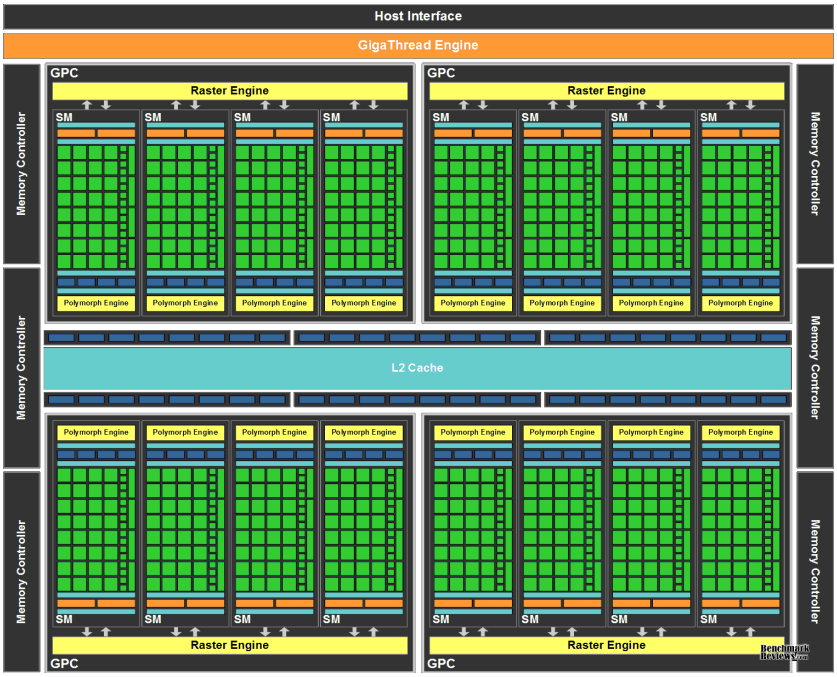


Figure 2.8: Diagram of the NVIDIA Fermi GPU Architecture[16].

November of 2010. The rest of the GPUs in this line of GPUs were released in the following months and the last ones being released in May of 2011[18].

This updated Fermi architecture were a notable update, with respects to performance and power management, compared to the original. Each streaming multiprocessor consists of up to 48 cores, each able to perform up to two floating point operations per clock cycle. Every streaming multiprocessor also contains up to eight special function units, each capable of up to four operations per clock cycle.

### 2.5.1 Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) is NVIDIAs hardware and software architecture that makes it possible for NVIDIA GPUs to execute programs that are not graphics programs[17]. CUDA provides both a low-level and a high-level API to interface with the GPU, as well as the C for

CUDA programming language to write kernels to be executed on a GPU[19].

A kernel written in C for CUDA is executed on a NVIDIA GPU by a set of threads. The threads are divided into groups, called thread blocks, and the thread blocks are organized in a grid.

A thread block executes on a single streaming multiprocessor and a thread runs on a core. All threads within a grid execute the same kernel.

When a thread block is scheduled for execution, its threads are divided into groups of 32 threads, called a warp, that execute the kernel concurrently.

### 2.5.2 OpenCL on NVIDIA GPUs

The CUDA architecture and the OpenCL architecture are quite similar[20]. A streaming multiprocessor equate an OpenCL compute unit and a CUDA thread correspond with an OpenCL work-item. Also, a CUDA thread block matches an OpenCL work-group.

When work-items are executed on a GPU, they are, as threads are in CUDA, divided into warps. Each work-group contains an integer number of warps. A work-group runs on a streaming multiprocessor and a work-item runs on a single core. All work-items within a warp execute the same instruction in lockstep. If the code path of work-items within a warp diverges, they are serialized. The number of work-items within a work-group should be divisible by the warp size for optimal performance.

OpenCL private memory maps to registers on the GPU. If a work-item needs more registers than is available, spill code is generated by the compiler and the extra memory needed is placed in a region of global memory.

Local memory in OpenCL maps to shared memory on the GPU. Avoiding bank conflicts is the key to achieve optimal performance when using local memory.

When work-items within a warp access global memory, the memory access is coalesced into as few memory transactions as possible. How many transactions that is issued is dependent on the size of the elements accessed and the memory access pattern of the work-items.

## 2.6 Previous Work

In GATLAS[21], they implement the BLAS routines gemm, gemv and saxpy in OpenCL. They use auto-tuning to find optimum kernels for AMD GPUs. [22] presents gemm kernels for AMD GPUs. These kernels are implemented in IL, the native, low-level, assembly-like language for programming AMD GPUs. In [23], they improve their previous gemm implementation in CUDA C and optimize it for the Fermi architecture. They do this by adding an additional level of blocking. Namely, they use register blocking in addition to shared memory blocking. [24] optimize a gemm kernel implemented in CUDA C for the Fermi architecture by using auto-tuning.

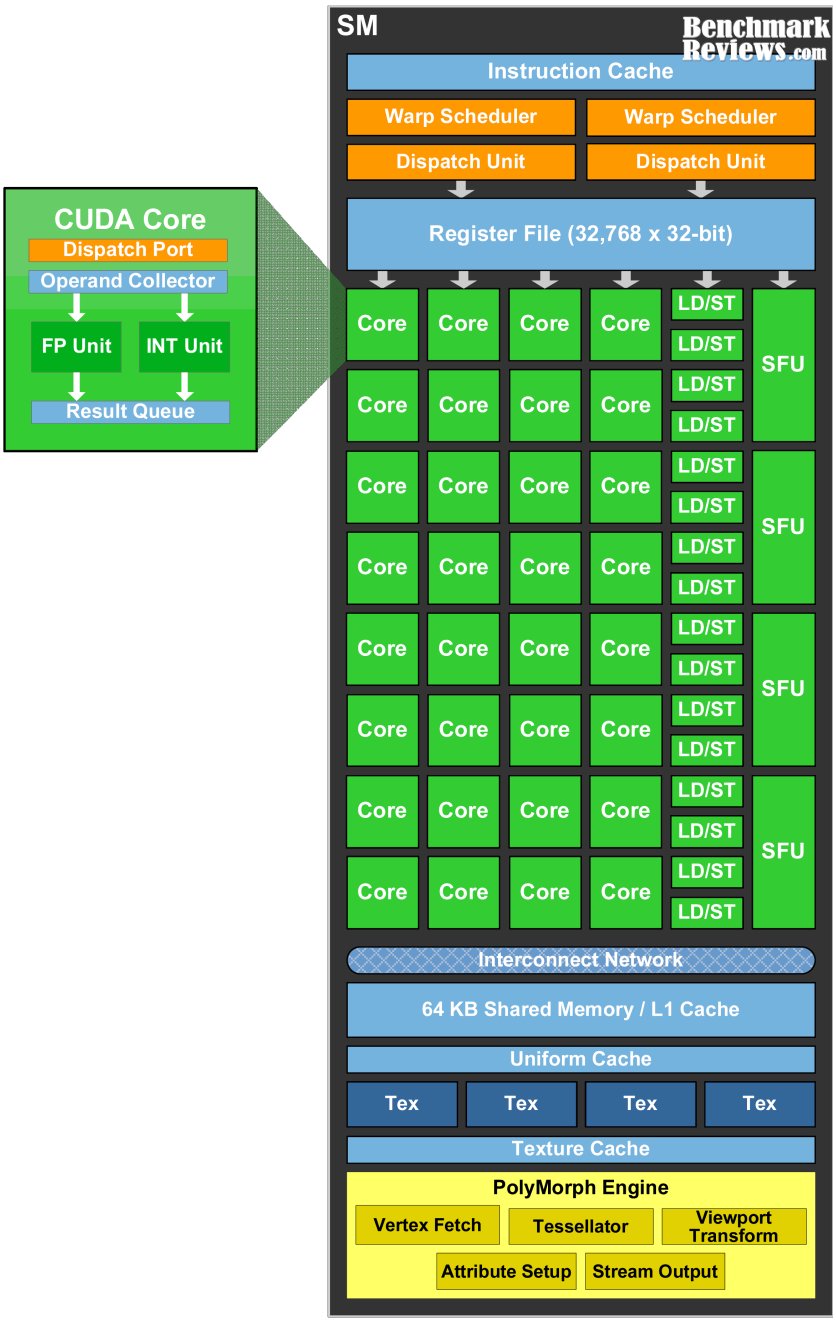


Figure 2.9: Diagram of a Fermi streaming multiprocessor and core[16].

# Chapter 3

# Implementation

In this chapter, we present the details of our implementation. First, we describe our auto-tuning framework in general. Then, we describe our code generator. Finally, we describe our search for the best kernel.

## 3.1 The Framework

Our auto-tuning framework were implemented in C++ and consisted of five classes in addition to the main function and some support functions related to initializing OpenCL and querying the system. See Figure 3.1 for a class diagram of the framework.

The Device class acted as an abstraction for an OpenCL device. It held all the information OpenCL provides for a device and methods to fetch them. The purpose of the class were to be used later to guide the code generator and by the auto-tuner, so that it would know which device to execute the kernels on. Additionally, it also contained functionality to output the device information for debugging purposes.

As the Device class were an abstraction for an OpenCL device, the Platform class were an abstraction for an OpenCL platform. It held the information provided by OpenCL for the platform and methods to retrieve that information. In addition, it also contained a list of the devices associated with it. The main purpose of the class were to hold the information about

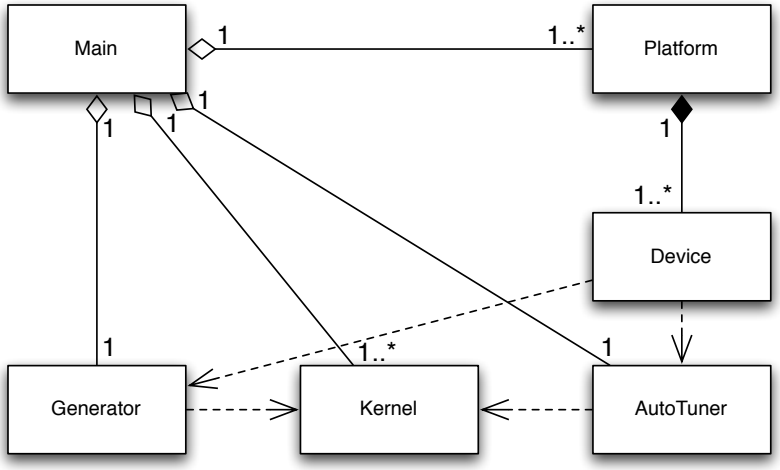


Figure 3.1: Class Diagram of the Auto-tuning Framework.

the OpenCL platforms and associated devices in the initial stages of the execution of the application, to allow the user to choose which device from which platform to generate kernels for.

The Kernel class were a wrapper for the kernels generated by the code generator. It contained the source code for a kernel that had been generated by the code generator, as well as the information required to execute the kernel on a device and methods to get this information when needed. More specifically, the information contained in the kernel class were the kernel name, the dimensions of the work-group to be used and the blocking size, needed to calculate the dimensions of the NDRange. Furthermore, the Kernel class contained functionality to output the kernel to a file and information about execution time on a device, used by the search.

The functionality related to the generation of kernels, were located in the Generator class. It contained one entry point method that took a Device object as input and returned a list of Kernel objects. Moreover, it contained the methods and supporting functions that generated the source code for this kernels. The methods that generated source code returned the source code in the form of a string that were then given as input when creating a Kernel object to be returned.

The AutoTuner class contained the functionality to perform the search for the best performing kernel. The search method took a Device object and a list of Kernel objects as input and gave one Kernel object as output. Additionally it contained supporting methods to allocate and initialize memory on the OpenCL device and functionality to reset the device if a kernel should, for some reason, crash or perform an illegal operation. It also contained functionality to test the validity of the output from the kernel for debugging purposed.

The remaining functionality of our framework consisted of functions to query the system for OpenCL platforms and devices and a function to allow the user to choose the device to generate a kernel for, in the event that several platforms and devices existed in the system.

The sequence of steps involved in our auto-tuning framework were the following:

1. Query system for platforms.
2. For all platforms: query for devices.
3. If user have started the program with the flag to choose a device, present the user with available devices. Otherwise, choose first device.
4. Send the device chosen as input to the code generator.
5. Send the list of kernels returned by the code generator as input to the search method.
6. Output the kernel returned by the search method to file.
7. Exit application.

## 3.2 The Code Generator

The code generator generated source code for five main versions of kernels. The difference between the different versions were the blocking scheme they employed. These versions consisted of:

1. No memory blocking.
2. Both matrix A and B blocked into local memory.
3. Matrix A blocked into shared memory and matrix B blocked into private memory.
4. Both matrix A and B blocked into private memory.
5. Both matrix A and B blocked into local memory. Both matrices blocked further into private memory.

Common to all versions were that the matrix C were divided into blocks in all of them, private memory were used to hold the temporary values before being written back to global memory and loops are unrolled. Each block were calculated by one work-group. Also, the generator assumed a row-major ordering of the matrices.

### 3.2.1 No Memory Blocking

This were the simplest version. The parameters used to generate the different kernel versions of this main version of kernels were the size of the block of the matrix C and the number of work-items to be used by each work-group. The number of work-items used per work-group had to be divisible by the number of elements in the block of the matrix C.

Depending on how many work-items were used to calculate one block of the matrix C, each work-item calculated one or more elements of C. For each element of C calculated, each work-item read one row from A and one column from B. See Figure 3.2 for a schematic of this version. The pseudocode for this version can be seen in Algorithm 1.

### 3.2.2 Both Matrices in Local Memory

The parameters to this version were, in addition to the blocking size and number of work-items to be used per work-group, if the blocks of local memory should be padded, to possibly avoid bank conflicts, and if they should be transposed when read from global to local memory.

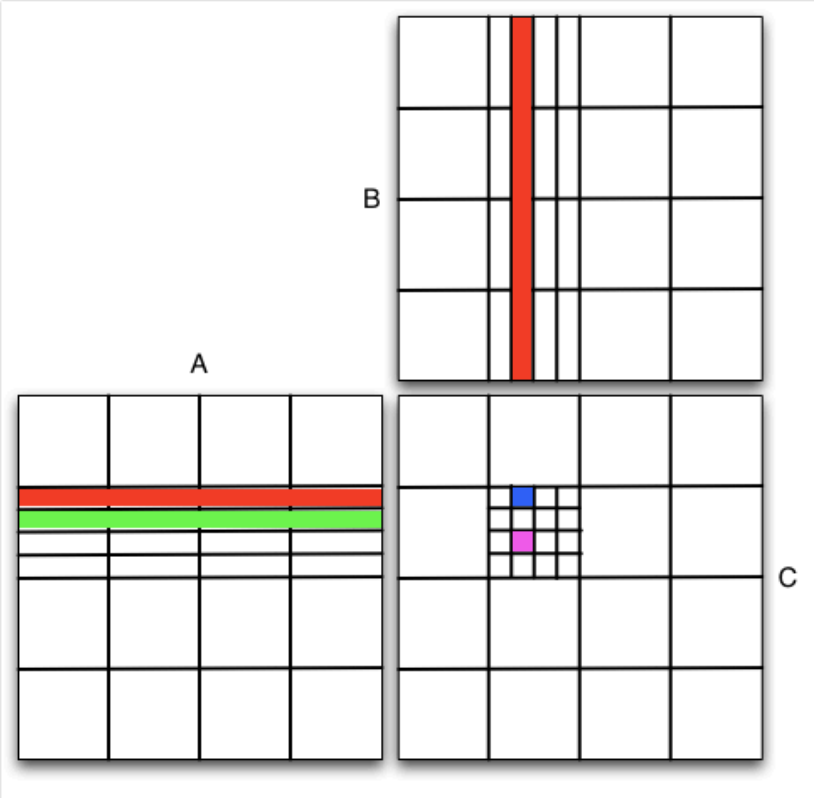


Figure 3.2: Schematic of no memory blocking version. In this reduced version, C is divided into blocks of 4x4 elements, which are computed by 4x2 work-items. Each work-item computes two elements. The work-item computing the blue and purple element of C reads the entire red and green row of A and the entire red column of B.

Each work-item calculated one or more elements of C, depending on how many work-items there were per block. For each block of matrix A and B, each work-item read one or more elements from global memory and placed them in local memory. Care was taken to ensure that reads from global memory were coalesced. Each work-item then read elements from local memory to calculate their respective elements of C. See Figure 3.3 for a schematic of this version. Also, the pseudocode for this version can be seen in Algorithm 2.

**Algorithm 1** Pseudocode of no memory blocking version.

```
Find your index in the index space
Move pointers of A, B and C to correct position
a_end ← a + k
Allocate private memory for elements of C
repeat
  for all Elements of C do
    c_element+ = a_element * b_element
  end for
  Increment pointers to A and B
until Pointer to A >= a_end
for all Elements of C do
  C = alpha * c_element + beta * C
end for
```

**Algorithm 2** Pseudocode of both matrices blocked in local memory version.

```
Find your index in the index space
Move pointers of A, B and C to correct position
a_end ← a + k
Allocate local memory for block of A and B
Allocate private memory for elements of C
repeat
  Read block of A into local memory
  Read block of B into local memory
  for all Elements of C do
    c_element+ = a_element * b_element
  end for
  Increment pointers to A and B
until Pointer to A >= a_end
for all Elements of C do
  C = alpha * c_element + beta * C
end for
```

### 3.2.3 One Matrix in Local Memory and One in Private Memory

For this version, the parameters were blocking size, register blocking size, number of work-items per work-group and if padding were to be used for the local memory.

Here, each work-item calculated one column of the block of the matrix C. Depending on the number of work-items in a work-group and the blocking size used for the matrix A, a work-item read one or more elements from A and placed it in local memory. Also here, care was taken so that reads from global memory were coalesced. Each work-item also read one or more elements from a column of B, depending on the register block size used. See Figure 3.4 for a schematic of this version and Algorithm 3 for the pseudocode.

**Algorithm 3** Pseudocode of one matrix blocked in local memory and one matrix blocked in private memory.

```
Find your index in the index space
Move pointers of A, B and C to correct position
a_end ← a + k
Allocate local memory for block of A
Allocate private memory for block of B
Allocate private memory for elements of C
repeat
  Read block of A into local memory
  for i = 1 → localmemoryblocksize/privatememoryblocksize do
    Read block of B into private memory
    for all Elements of C do
      c_element+ = a_element * b_element
    end for
    Increment pointer to B
  end for
  Increment pointer to A
until Pointer to A >= a_end
for all Elements of C do
  C = alpha * c_element + beta * C
end for
```

### 3.2.4 Both Matrices in Private Memory

In this version, the parameters were blocking size, register blocking size and number of work-items per work-group.

Depending on the number of work-items there were per block, each work-item calculated one or more elements of C. Each work-item read the register blocking size number of elements from B into private memory and depending on how many elements of C each work-item calculated, one of more times the register blocking size number of elements from A into private memory. A work-item, then read the register blocking size number of elements from the private memory block of A and B for each element of C it were to compute. Figure 3.5 contains a schematic of this version. See also Algorithm 4 for the pseudocode.

**Algorithm 4** Pseudocode of both matrices blocked in private memory.

```
Find your index in the index space
Move pointers of A, B and C to correct position
a_end ← a + k
Allocate private memory for blocks of A and B
Allocate private memory for elements of C
repeat
  Read block of A into private memory
  Read block of B into private memory
  for all Elements of C do
    c_element+ = a_element ∗ b_element
  end for
  Increment pointer to A
  Increment pointer to B
until Pointer to A >= a_end
for all Elements of C do
  C = alpha ∗ c_element + beta ∗ C
end for
```

### 3.2.5 Both Matrices in Local Memory and Private Memory

The parameters for this version were blocking size, register blocking size, number of work-items per work-group, if the blocks of local memory should be padded and if they should be transposed when read from global memory.

Each work-item calculated one or more elements of C, depending on how many work-items there were per block. Also depending on the number of work-items per block, a work-item read one or more elements of A and B from global memory and placed them in local memory. Care was taken to ensure that memory reads from global memory were coalesced. Then each work-item read register blocking size number of elements from the local memory blocks of A and B and placed them in the private memory blocks. Then, a work-item read the register blocking size number of elements from the private memory block of A and B for each element of C it were to compute. See Figure 3.6 for a schematic of this version and Algorithm 5 for the pseudocode.

## 3.3 The Search

In stead of generating all possible kernels and then prune the search tree afterwards, our code generator used the information about a device that OpenCL provides to minimize the number of kernels generated and to ensure that it only generated valid kernels. This reduced the search space to few enough kernels that a simple linear search were able to find the best kernel in a reasonable amount of time.

When all the kernels had been generated they were passed along to the search function, that compiled the kernel source code into OpenCL programs and created OpenCL kernel objects from them.

Each kernel were executed on the device a number of times, each execution being timed. Then the median time was chosen as the time for the kernel. This was done to ensure that no kernel was excluded for having the misfortune of being executed at the same time as the OpenCL runtime were performing some behind the scenes operation, causing the kernel execution time to be

**Algorithm 5** Pseudocode of both matrices blocked in both local memory and private memory.

```
Find your index in the index space
Move pointers of A, B and C to correct position
a_end ← a + k
Allocate local memory for block of A and B
Allocate private memory for block of A and B
Allocate private memory for elements of C
repeat
  Read block of A into local memory
  Read block of B into local memory
  for i = 1 → localmemoryblocksize/privatememoryblocksize do
    Read block of B into private memory
    for all Elements of C do
      Read block of A into private memory
      Increment pointer to local memory block of A
      c_element+ = a_element * b_element
    end for
    Increment pointer to local memory block of B
  end for
  Increment pointer to A
  Increment pointer to B
until Pointer to A >= a_end
for all Elements of C do
  C = alpha * c_element + beta * C
end for
```

higher than it normally would have been.

To measure the execution time of a kernel, we used the built in facilities in OpenCL to get profiling information from OpenCL commands. This way we could be sure that we had the most accurate timings of kernel executions that was possible.

See Algorithm 6 for the pseudo code for the search function.

**Algorithm 6** Pseudocode for search function.

```
Create memory buffers for matrix A, B and C
Copy initial data to memory buffers
Create a OpenCL CommandQueue
for all Kernel objects do
  Read kernel source code
  Create OpenCL program
  Compile OpenCL program
  Create OpenCL kernel
  Set kernel arguments
  for i = 1 → 5 do
    Submit kernel to CommandQueue
    Get timing results and add to list for this kernel
  end for
  Get the median timing result from list for this kernel
  if mediantime ≤ currentbesttime then
    Set best kernel to current kernel
  end if
end for
return best kernel
```

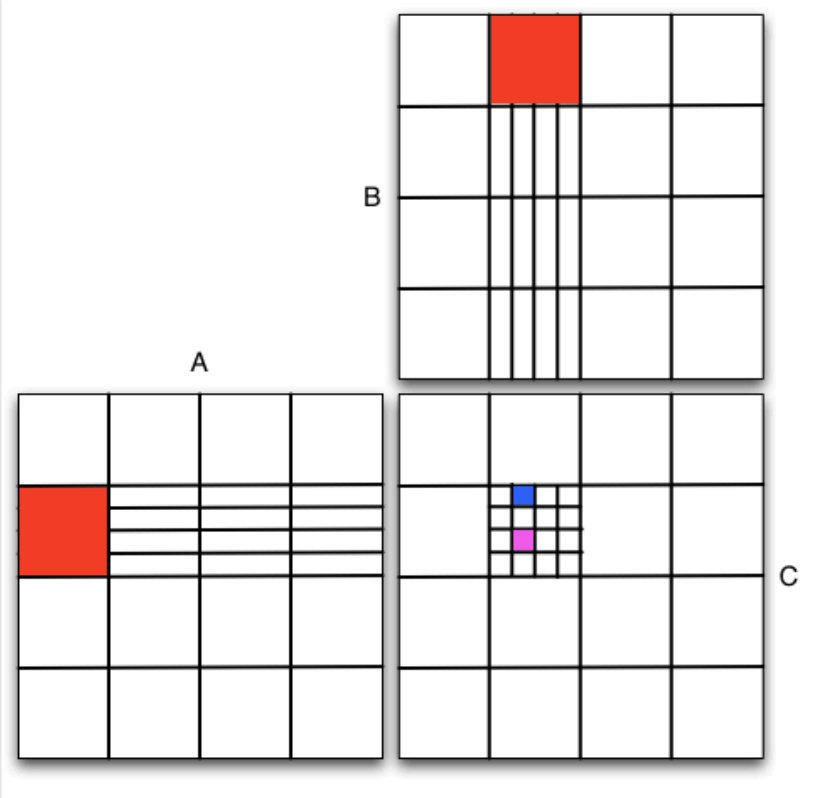


Figure 3.3: Schematic of local memory blocking version. In this reduced version, C is divided into blocks of 4x4 elements, which are computed by 4x2 work-items. Each work-item computes two elements, e.g. the blue and purple. The red blocks in A and B are the blocks that are read into local memory. Each work-item read two elements of A and two elements of B and places them in local memory. Each work-item then reads one row from the local memory block of A and one column of the local memory block of B for each element of C it computes.

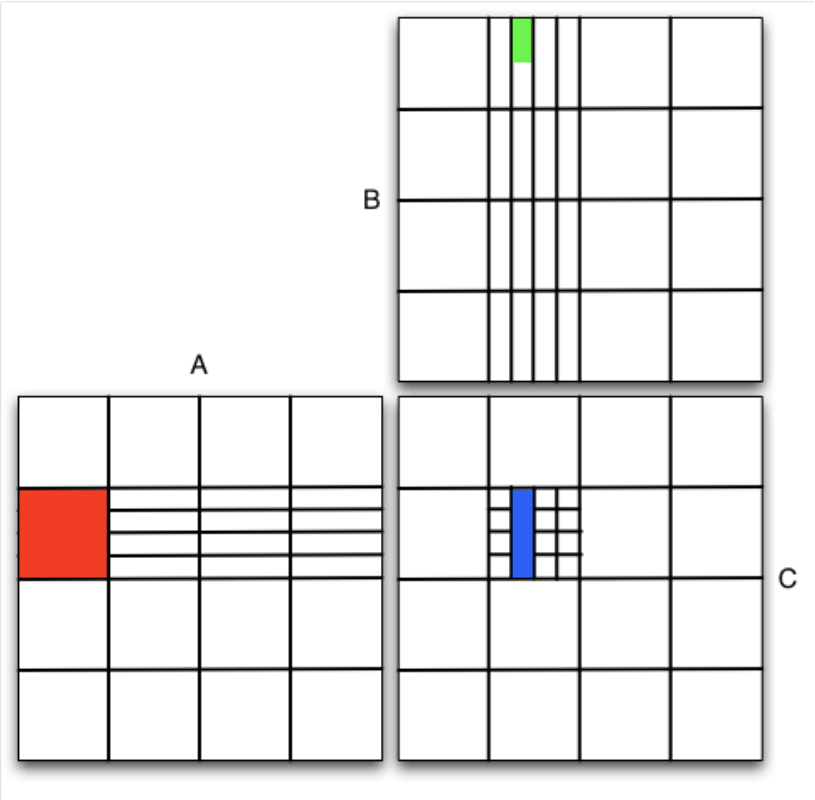


Figure 3.4: Schematic of one matrix blocked in local memory and one matrix blocked in private memory. In this reduced version, C is divided into blocks of 4x4 elements, which are computed by 4 work-items. Each work-item computes one column of the block of C, e.g. the blue column. The red block of A is a block that is read into local memory and the green block of B is a block that is read into private memory. Each work-item reads two elements of A and places them in local memory and reads two elements of B and places them in private memory. Each work-item then reads one row from the local memory block of A and the elements in private memory for the block of B for each element of C it computes.

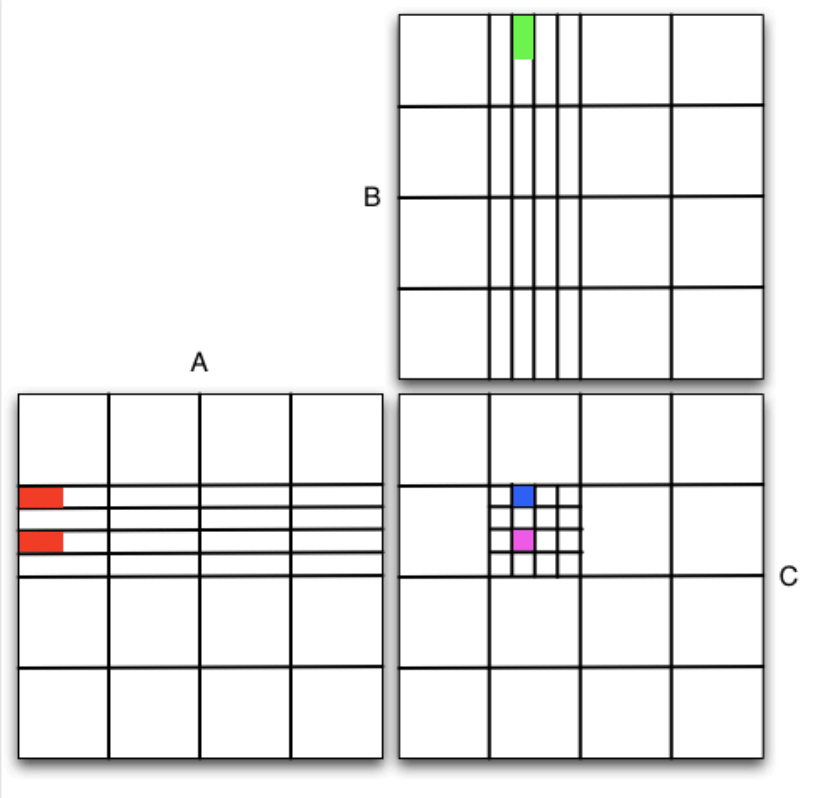


Figure 3.5: Schematic of both matrices blocked in private memory. In this reduced version, C is divided into blocks of 4x4 elements, which are computed by 4x2 work-items. Each work-item computes two elements of C, e.g the blue and purple. The two red blocks of A and the green block of B are read into private memory. Each work-item then reads the elements of one of the blocks of A from private memory and the elements of the block of B from private memory. The same is done to calculate the other element of C, only the elements of the other block of A is read from private memory instead.

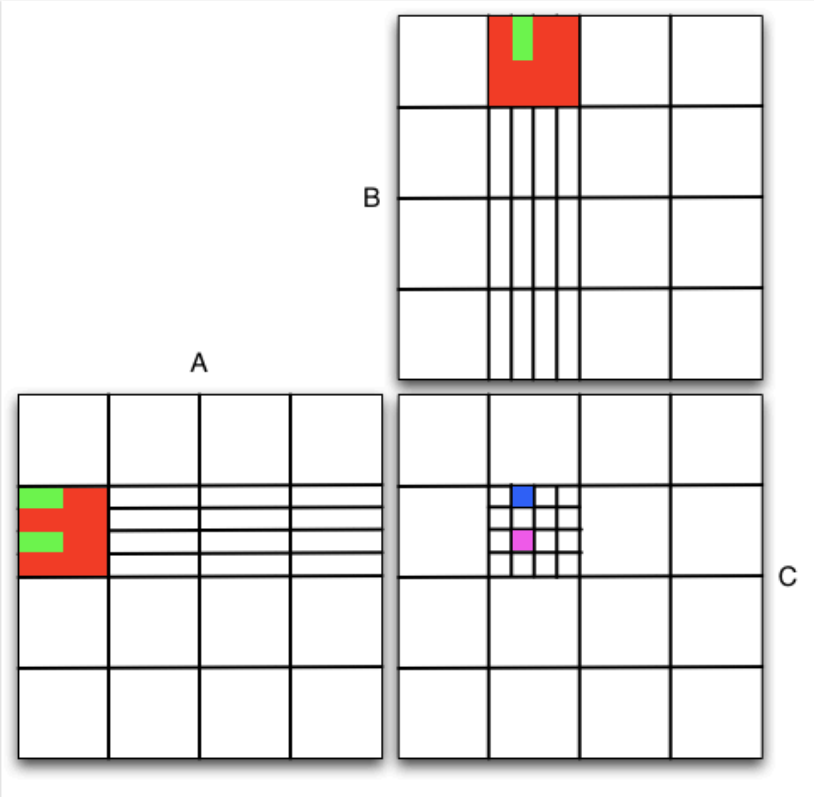


Figure 3.6: Schematic of both matrices blocked in local memory and in private memory. In this reduced version, C is divided into blocks of 4x4 elements, which are computed by 4x2 work-items. Each work-item computes two elements of C, e.g the blue and purple. The red blocks of A and B are the blocks that are read into local memory. Then each-work item read the green blocks of A and B from their local memory block into private memory, before they read the elements of one of the green blocks of A and the green block of B to compute the first element of C. The same is repeated for the second element, only the second green block of A is read instead

# Chapter 4

# Benchmarks and Results

In this chapter, we test out program on different GPUs. We start with presenting the different system configurations that we performed the tests on. Finally we present the results from running the generated kernels on different matrix sizes.

## 4.1 System Configurations

We have run our program on three different GPUs, AMD HD5750, AMD HD5870 and NVIDIA C2070. The configuration for these three systems can be seen in Table 4.1, Table 4.2 and Table 4.3 respectively.

| System 1 | |
|---|---|
| Processor | Intel core i5 @ 2,8GHz |
| Memory | 8GB |
| GPU | AMD HD5750 |
| Driver version | 8.812.0.0 |
| Operating System | Windows 7 (64bit) |

Table 4.1: Table describing the system with the AMD HD5750 GPU.

| System 2 | |
|---|---|
| Processor | Intel core i5 @ 2,67GHz |
| Memory | 4GB |
| GPU | AMD HD5870 |
| Driver version | 8.821.0.0 |
| Operating System | Windows 7 (64bit) |

Table 4.2: Table describing the system with the AMD HD5870 GPU.

| System 3 | |
|---|---|
| Processor | Intel core i7 @ 3,2GHz |
| Memory | 24GB |
| GPU | NVIDIA C2070 |
| Driver version | CUDA Toolkit 3.2.16 |
| Operating System | Ubuntu 10.04 LTS (64bit) |

Table 4.3: Table describing the system with the NVIDIA C2070 GPU.

## 4.2 Benchmarks

We ran our program on the three systems described in Section 4.1 and executed the kernel that was output with different matrix sizes containing floats as input.

For comparison with the results from our program, we have used a few different libraries. The first used for comparison were ViennaCL[25]. ViennaCL is a full implementation of the BLAS API in OpenCL, capable of running on all OpenCL devices.

The second library we used for comparison were the native library supplied from the vendor of the respective GPU. For the AMD GPUs this were ACML-GPU[26] and for the NVIDIA GPU this were CUBLAS[27].

As we were unable to compile the ACML-GPU on the system with the AMD HD5870 GPU, results from this library is not provided. The ACML-GPU is only supported with 64-bit operating systems and although the system with the AMD HD5870 GPU were running Windows 7 64-bit edition, there was some 32-bit/64-bit compatibility issues when we tried to compile

the library. We were never able to resolve this issue in the allotted time.

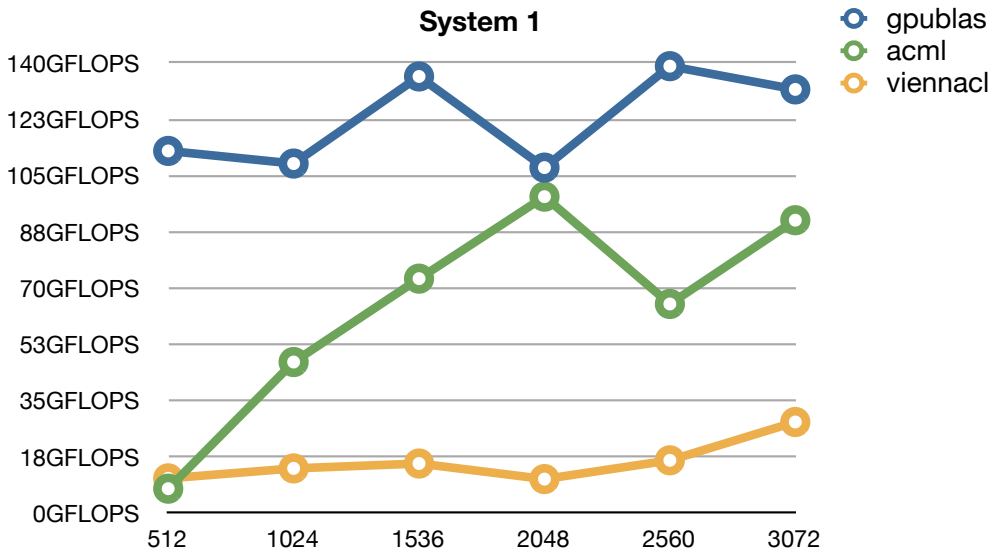


Figure 4.1: Graph of executions with different matrix sizes on the AMD HD5750 GPU.

As you can see from Figure 4.1, our program achieved a speedup of between 1,09 and 14,67 over ACML-GPU. The lowest speed up were achieved at the matrix size of 2048, where the ACML-GPU had its peek and our generated version had a low-point. For the other matrix sizes the speedup were roughly between 1,5 and 2. Our version also performed considerably better than ViennaCL.

Figure 4.2 gives the result of running our generated version and ViennaCL on the AMD HD5870 GPU. From looking at the pattern of the graph for our version, one can see that it showed similar characteristics to the one in Figure 4.1.

As can be seen from Figure 4.3, our generated kernel showed different characteristics on the NVIDIA GPU than it did on the GPUs from AMD. It quickly came up to a performance of approximately 260GFLOPS. Compared to CUBLAS, our generated kernel had between 60% and 97% of the performance. Our kernel had 60% of CUBLAS' performance at the matrix

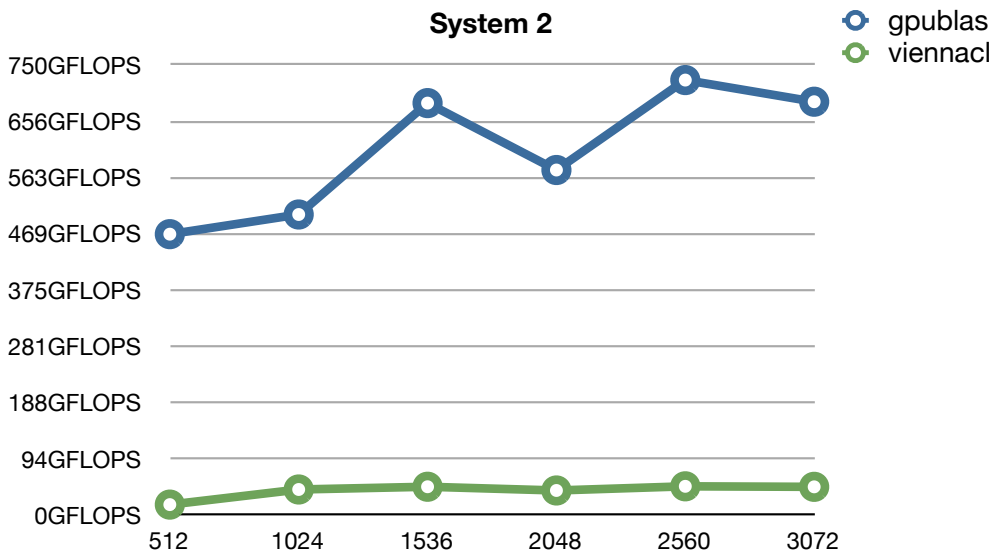


Figure 4.2: Graph of executions with different matrix sizes on the AMD HD5870 GPU.

size when CUBLAS had a spike and had 97% of CUBLAS' performance with the matrix size of 512. For the rest of the matrix sizes though, our generated kernel performed at approximately 85% of CUBLAS. Also on the NVIDIA GPU, our generated kernel performed considerably better than ViennaCL.

The time it took to generate and find the best kernels varied among the different GPUs. See Table 4.4.

| System | Time |
|---|---|
| System 1 | 148s |
| System 2 | 96s |
| System 3 | 3017s |

Table 4.4: Table listing the time it took to generate and search for the best kernel on the different systems.

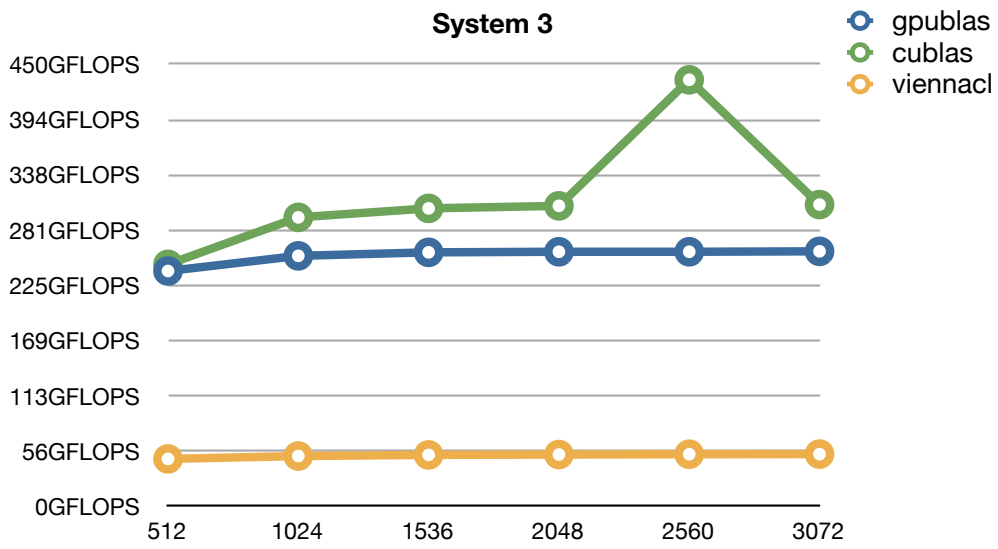


Figure 4.3: Graph of executions with different matrix sizes on the NVIDIA C2070 GPU.

# Chapter 5

# Conclusion

We have, in this thesis, implemented an auto-tuning framework that generates source code for OpenCL kernels, intended to be executed on a GPU. We use information about the GPU intended to run the kernels generated that is provided by OpenCL to guide the code generator in its choice of parameter values. When the kernels have been generated our auto-tuning framework searches for the best kernel and outputs it to file.

Our code generator is able to generate kernels that uses a vast number of blocking schemes and several blocking sizes. Along with differing number of work-items and arrangement of work-items, this gives enough different possible kernel variants that it should be able to find good performing kernels for most GPUs.

As can be seen from Section 4.2, the performance of the kernels chosen by our auto-tuning framework vary depending on which vendors GPUs our program is run on. Compared to the native libraries from AMD and NVIDIA our auto-tuning framework performs better on AMD GPUs than it does on GPUs from NVIDIA.

The reasons for this can be one of several. It could be that our generator as it is now, generates kernels that map better to the AMD GPU architecture than it does the NVIDIA GPU architecture. It could also be that as NVIDIA have had GPUs capable of executing programs other than graphics programs for longer than AMD have, the native libraries supplied by NVIDIA

is more mature than the ones supplied by AMD, making it easier to achieve good performance compared to the AMD library than it is compared to the NVIDIA library. It could also be some sort of combination of the two.

Table 4.4 shows the time it took to generate and find the best kernel on the different GPUs. As you can see, the time it took on the NVIDIA GPU vastly exceeds the time it took on the AMD GPUs. This is due to some strange behavior that we observed when running our auto-tuning framework on the NVIDIA GPU. For some reason, when trying to compile some of the kernels on the NVIDIA GPU the compiler fails and returns an undefined error code[1].Whenever this undefined error occur, the compilation stage of the search method takes considerably more time than it does when the kernels compiles without error.

This error occurs for a large number of kernels and accounts for most of the time difference between the AMD and NVIDIA GPUs for generating and finding the best kernel. As all the kernels that does not compile on the NVIDIA GPU compiles and execute without problems on the AMD GPUs, we believe that this is a bug in NVIDIAs current OpenCL compiler. However, further investigations are needed.

Our goal for this thesis were to create an auto-tuning framework that generated OpenCL kernels that performed matrix multiplication on GPUs and had good performance on different GPUs. As we achieve better performance than the currently available OpenCL BLAS library and also what we think is good performance compared to he native vendor supplied BLAS libraries, we consider this goal to be achieved.

## 5.1 Future Work

Currently our code generator only generates kernels for matrix multiplication where the matrices A and B are in their original form. To expand its usability, the code generator should be expanded to generate kernels for the versions of matrix multiplication where A or B or both are transposed as well.

[1]By undefined error code, we mean an error code that is not specified in the OpenCL Specification.

The kernels generated by our code generator is at the moment only able to handle matrices of sizes that are multiples of the blocking size employed. This is easily circumvented by padding the matrices with zeroes. Whether this is the best strategy or if some sort of clean-up code should be used instead is something that needs further investigation.

Matrix multiplication is only one of many functions in BLAS. We believe that it is important to have a BLAS library in OpenCL that achieves good performance on GPUs from multiple vendors is important to ensure that OpenCL becomes the preferred framework for programming GPUs and is not put in the shadow of native, vendor specific alternatives like CUDA. Therefore, expansion of the auto-tuning framework to generate and find optimal kernels for other BLAS routines is also warranted.

# Bibliography


[1] R. Clint Whaley, Antoine Petitet, Jack J. Dongarra Automated Empirical Optimization of Software and the ATLAS Project http://www.netlib.org, September 19, 2000

[2] Wikipedia, BLAS [web] http://en.wikipedia.org/w/index.php?title=Basic_Linear_Algebra_Subprograms&oldid=420631241 [cited at 19. June 2011]

[3] netlib.org BLAS FAQ [web] http://www.netlib.org/blas/faq.html [cited at 19. June 2011]

[4] netlib.org BLAS [web] http://www.netlib.org/blas/ [cited at 19. June 2011]

[5] Wikipedia, OpenCL [web] http://en.wikipedia.org/w/index.php?title=OpenCL&oldid=434928181 [cited at 19. June 2011]

[6] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, Satoru Tagawa. The OpenCL Programming Book, Fixstars Corporation, 2010.

[7] Khronos Group, The OpenCL Specification Version: 1.0 [Web] http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf [cited at 19. June 2011]

[8] Jens Breitbart, Claudia Fohry OpenCL - An effective programming model for data parallel computations at the Cell Broadband Engine Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on 19-23 April 2010

[9] Benedict R. Gaster OpenCL Overview [web] http://www.khronos.org/developers/library/overview/opencl_overview.pdf [cited at 19. June 2011]

[10] Wikipedia, Evergreen (GPU family) [web] http://en.wikipedia.org/w/index.php?title=Evergreen_(GPU_family) &oldid=434320559 [cited at 19. June 2011]

[11] AMD Accelerated Parallel Processing OpenCL Programming Guide [web] [cited at 19. June 2011]

[12] Chris Angelini AMD Radeon HD 6990 4 GB Review: Antilles Makes (Too Much) Noise [web] http://www.tomshardware.com/reviews/radeon-hd-6990-antilles-crossfire,2878.html [cited at 19. June 2011]

[13] Chris Angelini Building Cayman By Improving Cypress [web] http://www.tomshardware.com/reviews/radeon-hd-6970-radeon-hd-6950-cayman,2818-2.html [cited at 19. June 2011]

[14] Wikipedia, Northern Islands (GPU family) [web] http://en.wikipedia.org/w/index.php?title=Northern_Islands_(GPU_family) &oldid=435021695 [cited at 19. June 2011]

[15] Wikipedia, GeForce 400 Series [web] http://en.wikipedia.org/w/index.php?title=GeForce_400_Series &oldid=432077009 [cited at 19. June 2011]

[16] Olin Coles NVIDIA GF100 GPU Fermi Graphics Architecture [web] http://benchmarkreviews.com/index.php?option=com_content &task=view&id=440&Itemid=63 [cited at 19. June 2011]

[17] NVIDIA, Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi [web] http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIAFermiComputeArchitectureWhitepaper.pdf [cited on 19. June 2011]

[18] Wikipedia, GeForce 500 Series [web] http://en.wikipedia.org/w/index.php?title=GeForce_500_Series &oldid=434673700 [cited on 19. June 2011]

[19] Wikipedia, CUDA [web] http://en.wikipedia.org/w/index.php?title=CUDA &oldid=435074425 [cited on 19. June 2011]

[20] NVIDIA, OpenCL Programming Guide for the CUDA Architecture [web] http://developer.download.nvidia.com/compute/cuda/3_2_prod/ toolkit/docs/OpenCL_Programming_Guide.pdf [cited on 19. June 2011]

[21] Chris Jang GATLAS GPU Automatically Tuned Linear Algebra Software [web] http://golem5.org/gatlas/ [cited on 19. June 2011]

[22] Naohito Nakasato A Fast GEMM Implementation On a Cypress GPU ACM SIGMETRICS Performance Evaluation Review - Special issue on the 1st international workshop on performance modeling, benchmarking and simulation of high performance computing systems (PMBS 10), Volume 38 Issue 4, March 2011

[23] Rajib Nath, Stanimire Tomov, Jack Dongarra An Improved MAGMA GEMM for Fermi GPUs http://www.netlib.org, July 10, 2010

[24] Xiang Cui, Yifeng Chen, Changyou Zhang, Hong Mei Auto-tuning Dense Matrix Multiplication for GPGPU with Cache Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on 8-10 December 2010, p.237-242

[25] ViennaCL [web] http://viennacl.sourceforge.net/ [cited on 19. June 2011]

[26] AMD Core Math Library for Graphic Processors (ACML-GPU) http://developer.amd.com/libraries/acmlgpu/Pages/default.aspx [cited on 19. June 2011]

[27] NVIDIA, CUDA CUBLAS Library [web] http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf [cited on 19. June 2011]

# Appendix A

# Selected Source Code

We will here present the source code of the kernels generated for the different GPUs. Interestingly enough, the auto-tuning framework chose the same kernel for both the AMD HD5750 and the AMD HD5870. The source code for the AMD GPUs are presented in A.1 and the source code for the NVIDIA GPU is presented in A.2. The rest of the source code for the auto-tuning framework is available upon request.

```
1  __kernel __attribute__((reqd_work_group_size
       (16, 4, 1)))
2  void gemm(__global float *a, __global float *b,
       __global float *c,
3  int m, int n, int k,
4  float alpha, float beta)
5  {
6  const int bidx = get_group_id(0);
7  const int bidy = get_group_id(1);
8  const int idx = get_local_id(0);
9  const int idy = get_local_id(1);
10 a += (bidy * 16 + idy) * k + idx;
11 b += bidx * 64 + idx + idy * 16;
12 c += bidx * 64 + bidy * 16 * n + idx + idy *
       16;
```

```
13          __global float *a_end = a + k;
14          __local float block_a[16][16+1];
15          float c_reg[16] = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f
               , 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
               0.0f, 0.0f, 0.0f, 0.0f};
16          do {
17          float b_reg[4] = {b[0 * n], b[1 * n], b[2 * n],
                b[3 * n]};
18          block_a[idy + 0 * 4][idx] = a[0 * 4 * k];
19          block_a[idy + 1 * 4][idx] = a[1 * 4 * k];
20          block_a[idy + 2 * 4][idx] = a[2 * 4 * k];
21          block_a[idy + 3 * 4][idx] = a[3 * 4 * k];
22          barrier(CLK_LOCAL_MEM_FENCE);
23          b += 4*n;
24          c_reg[0] += block_a[0][0] * b_reg[0];
25          c_reg[1] += block_a[0][1] * b_reg[0];
26          c_reg[2] += block_a[0][2] * b_reg[0];
27          c_reg[3] += block_a[0][3] * b_reg[0];
28          c_reg[4] += block_a[0][4] * b_reg[0];
29          c_reg[5] += block_a[0][5] * b_reg[0];
30          c_reg[6] += block_a[0][6] * b_reg[0];
31          c_reg[7] += block_a[0][7] * b_reg[0];
32          c_reg[8] += block_a[0][8] * b_reg[0];
33          c_reg[9] += block_a[0][9] * b_reg[0];
34          c_reg[10] += block_a[0][10] * b_reg[0];
35          c_reg[11] += block_a[0][11] * b_reg[0];
36          c_reg[12] += block_a[0][12] * b_reg[0];
37          c_reg[13] += block_a[0][13] * b_reg[0];
38          c_reg[14] += block_a[0][14] * b_reg[0];
39          c_reg[15] += block_a[0][15] * b_reg[0];
40          b_reg[0] = b[0 * n];
41          c_reg[0] += block_a[1][0] * b_reg[1];
42          c_reg[1] += block_a[1][1] * b_reg[1];
```

```
43          c_reg[2] += block_a[1][2] * b_reg[1];
44          c_reg[3] += block_a[1][3] * b_reg[1];
45          c_reg[4] += block_a[1][4] * b_reg[1];
46          c_reg[5] += block_a[1][5] * b_reg[1];
47          c_reg[6] += block_a[1][6] * b_reg[1];
48          c_reg[7] += block_a[1][7] * b_reg[1];
49          c_reg[8] += block_a[1][8] * b_reg[1];
50          c_reg[9] += block_a[1][9] * b_reg[1];
51          c_reg[10] += block_a[1][10] * b_reg[1];
52          c_reg[11] += block_a[1][11] * b_reg[1];
53          c_reg[12] += block_a[1][12] * b_reg[1];
54          c_reg[13] += block_a[1][13] * b_reg[1];
55          c_reg[14] += block_a[1][14] * b_reg[1];
56          c_reg[15] += block_a[1][15] * b_reg[1];
57          b_reg[1] = b[1 * n];
58          c_reg[0] += block_a[2][0] * b_reg[2];
59          c_reg[1] += block_a[2][1] * b_reg[2];
60          c_reg[2] += block_a[2][2] * b_reg[2];
61          c_reg[3] += block_a[2][3] * b_reg[2];
62          c_reg[4] += block_a[2][4] * b_reg[2];
63          c_reg[5] += block_a[2][5] * b_reg[2];
64          c_reg[6] += block_a[2][6] * b_reg[2];
65          c_reg[7] += block_a[2][7] * b_reg[2];
66          c_reg[8] += block_a[2][8] * b_reg[2];
67          c_reg[9] += block_a[2][9] * b_reg[2];
68          c_reg[10] += block_a[2][10] * b_reg[2];
69          c_reg[11] += block_a[2][11] * b_reg[2];
70          c_reg[12] += block_a[2][12] * b_reg[2];
71          c_reg[13] += block_a[2][13] * b_reg[2];
72          c_reg[14] += block_a[2][14] * b_reg[2];
73          c_reg[15] += block_a[2][15] * b_reg[2];
74          b_reg[2] = b[2 * n];
75          c_reg[0] += block_a[3][0] * b_reg[3];
```

```
76          c_reg[1] += block_a[3][1] * b_reg[3];
77          c_reg[2] += block_a[3][2] * b_reg[3];
78          c_reg[3] += block_a[3][3] * b_reg[3];
79          c_reg[4] += block_a[3][4] * b_reg[3];
80          c_reg[5] += block_a[3][5] * b_reg[3];
81          c_reg[6] += block_a[3][6] * b_reg[3];
82          c_reg[7] += block_a[3][7] * b_reg[3];
83          c_reg[8] += block_a[3][8] * b_reg[3];
84          c_reg[9] += block_a[3][9] * b_reg[3];
85          c_reg[10] += block_a[3][10] * b_reg[3];
86          c_reg[11] += block_a[3][11] * b_reg[3];
87          c_reg[12] += block_a[3][12] * b_reg[3];
88          c_reg[13] += block_a[3][13] * b_reg[3];
89          c_reg[14] += block_a[3][14] * b_reg[3];
90          c_reg[15] += block_a[3][15] * b_reg[3];
91          b_reg[3] = b[3 * n];
92          b += 4*n;
93          c_reg[0] += block_a[0][0] * b_reg[0];
94          c_reg[1] += block_a[0][1] * b_reg[0];
95          c_reg[2] += block_a[0][2] * b_reg[0];
96          c_reg[3] += block_a[0][3] * b_reg[0];
97          c_reg[4] += block_a[0][4] * b_reg[0];
98          c_reg[5] += block_a[0][5] * b_reg[0];
99          c_reg[6] += block_a[0][6] * b_reg[0];
100         c_reg[7] += block_a[0][7] * b_reg[0];
101         c_reg[8] += block_a[0][8] * b_reg[0];
102         c_reg[9] += block_a[0][9] * b_reg[0];
103         c_reg[10] += block_a[0][10] * b_reg[0];
104         c_reg[11] += block_a[0][11] * b_reg[0];
105         c_reg[12] += block_a[0][12] * b_reg[0];
106         c_reg[13] += block_a[0][13] * b_reg[0];
107         c_reg[14] += block_a[0][14] * b_reg[0];
108         c_reg[15] += block_a[0][15] * b_reg[0];
```

```
109             b_reg[0] = b[0 * n];
110             c_reg[0] += block_a[1][0] * b_reg[1];
111             c_reg[1] += block_a[1][1] * b_reg[1];
112             c_reg[2] += block_a[1][2] * b_reg[1];
113             c_reg[3] += block_a[1][3] * b_reg[1];
114             c_reg[4] += block_a[1][4] * b_reg[1];
115             c_reg[5] += block_a[1][5] * b_reg[1];
116             c_reg[6] += block_a[1][6] * b_reg[1];
117             c_reg[7] += block_a[1][7] * b_reg[1];
118             c_reg[8] += block_a[1][8] * b_reg[1];
119             c_reg[9] += block_a[1][9] * b_reg[1];
120             c_reg[10] += block_a[1][10] * b_reg[1];
121             c_reg[11] += block_a[1][11] * b_reg[1];
122             c_reg[12] += block_a[1][12] * b_reg[1];
123             c_reg[13] += block_a[1][13] * b_reg[1];
124             c_reg[14] += block_a[1][14] * b_reg[1];
125             c_reg[15] += block_a[1][15] * b_reg[1];
126             b_reg[1] = b[1 * n];
127             c_reg[0] += block_a[2][0] * b_reg[2];
128             c_reg[1] += block_a[2][1] * b_reg[2];
129             c_reg[2] += block_a[2][2] * b_reg[2];
130             c_reg[3] += block_a[2][3] * b_reg[2];
131             c_reg[4] += block_a[2][4] * b_reg[2];
132             c_reg[5] += block_a[2][5] * b_reg[2];
133             c_reg[6] += block_a[2][6] * b_reg[2];
134             c_reg[7] += block_a[2][7] * b_reg[2];
135             c_reg[8] += block_a[2][8] * b_reg[2];
136             c_reg[9] += block_a[2][9] * b_reg[2];
137             c_reg[10] += block_a[2][10] * b_reg[2];
138             c_reg[11] += block_a[2][11] * b_reg[2];
139             c_reg[12] += block_a[2][12] * b_reg[2];
140             c_reg[13] += block_a[2][13] * b_reg[2];
141             c_reg[14] += block_a[2][14] * b_reg[2];
```

```
142             c_reg[15] += block_a[2][15] * b_reg[2];
143             b_reg[2] = b[2 * n];
144             c_reg[0] += block_a[3][0] * b_reg[3];
145             c_reg[1] += block_a[3][1] * b_reg[3];
146             c_reg[2] += block_a[3][2] * b_reg[3];
147             c_reg[3] += block_a[3][3] * b_reg[3];
148             c_reg[4] += block_a[3][4] * b_reg[3];
149             c_reg[5] += block_a[3][5] * b_reg[3];
150             c_reg[6] += block_a[3][6] * b_reg[3];
151             c_reg[7] += block_a[3][7] * b_reg[3];
152             c_reg[8] += block_a[3][8] * b_reg[3];
153             c_reg[9] += block_a[3][9] * b_reg[3];
154             c_reg[10] += block_a[3][10] * b_reg[3];
155             c_reg[11] += block_a[3][11] * b_reg[3];
156             c_reg[12] += block_a[3][12] * b_reg[3];
157             c_reg[13] += block_a[3][13] * b_reg[3];
158             c_reg[14] += block_a[3][14] * b_reg[3];
159             c_reg[15] += block_a[3][15] * b_reg[3];
160             b_reg[3] = b[3 * n];
161             b += 4*n;
162             c_reg[0] += block_a[0][0] * b_reg[0];
163             c_reg[1] += block_a[0][1] * b_reg[0];
164             c_reg[2] += block_a[0][2] * b_reg[0];
165             c_reg[3] += block_a[0][3] * b_reg[0];
166             c_reg[4] += block_a[0][4] * b_reg[0];
167             c_reg[5] += block_a[0][5] * b_reg[0];
168             c_reg[6] += block_a[0][6] * b_reg[0];
169             c_reg[7] += block_a[0][7] * b_reg[0];
170             c_reg[8] += block_a[0][8] * b_reg[0];
171             c_reg[9] += block_a[0][9] * b_reg[0];
172             c_reg[10] += block_a[0][10] * b_reg[0];
173             c_reg[11] += block_a[0][11] * b_reg[0];
174             c_reg[12] += block_a[0][12] * b_reg[0];
```

```
175         c_reg[13] += block_a[0][13] * b_reg[0];
176         c_reg[14] += block_a[0][14] * b_reg[0];
177         c_reg[15] += block_a[0][15] * b_reg[0];
178         b_reg[0] = b[0 * n];
179         c_reg[0] += block_a[1][0] * b_reg[1];
180         c_reg[1] += block_a[1][1] * b_reg[1];
181         c_reg[2] += block_a[1][2] * b_reg[1];
182         c_reg[3] += block_a[1][3] * b_reg[1];
183         c_reg[4] += block_a[1][4] * b_reg[1];
184         c_reg[5] += block_a[1][5] * b_reg[1];
185         c_reg[6] += block_a[1][6] * b_reg[1];
186         c_reg[7] += block_a[1][7] * b_reg[1];
187         c_reg[8] += block_a[1][8] * b_reg[1];
188         c_reg[9] += block_a[1][9] * b_reg[1];
189         c_reg[10] += block_a[1][10] * b_reg[1];
190         c_reg[11] += block_a[1][11] * b_reg[1];
191         c_reg[12] += block_a[1][12] * b_reg[1];
192         c_reg[13] += block_a[1][13] * b_reg[1];
193         c_reg[14] += block_a[1][14] * b_reg[1];
194         c_reg[15] += block_a[1][15] * b_reg[1];
195         b_reg[1] = b[1 * n];
196         c_reg[0] += block_a[2][0] * b_reg[2];
197         c_reg[1] += block_a[2][1] * b_reg[2];
198         c_reg[2] += block_a[2][2] * b_reg[2];
199         c_reg[3] += block_a[2][3] * b_reg[2];
200         c_reg[4] += block_a[2][4] * b_reg[2];
201         c_reg[5] += block_a[2][5] * b_reg[2];
202         c_reg[6] += block_a[2][6] * b_reg[2];
203         c_reg[7] += block_a[2][7] * b_reg[2];
204         c_reg[8] += block_a[2][8] * b_reg[2];
205         c_reg[9] += block_a[2][9] * b_reg[2];
206         c_reg[10] += block_a[2][10] * b_reg[2];
207         c_reg[11] += block_a[2][11] * b_reg[2];
```

```
208             c_reg[12] += block_a[2][12] * b_reg[2];
209             c_reg[13] += block_a[2][13] * b_reg[2];
210             c_reg[14] += block_a[2][14] * b_reg[2];
211             c_reg[15] += block_a[2][15] * b_reg[2];
212             b_reg[2] = b[2 * n];
213             c_reg[0] += block_a[3][0] * b_reg[3];
214             c_reg[1] += block_a[3][1] * b_reg[3];
215             c_reg[2] += block_a[3][2] * b_reg[3];
216             c_reg[3] += block_a[3][3] * b_reg[3];
217             c_reg[4] += block_a[3][4] * b_reg[3];
218             c_reg[5] += block_a[3][5] * b_reg[3];
219             c_reg[6] += block_a[3][6] * b_reg[3];
220             c_reg[7] += block_a[3][7] * b_reg[3];
221             c_reg[8] += block_a[3][8] * b_reg[3];
222             c_reg[9] += block_a[3][9] * b_reg[3];
223             c_reg[10] += block_a[3][10] * b_reg[3];
224             c_reg[11] += block_a[3][11] * b_reg[3];
225             c_reg[12] += block_a[3][12] * b_reg[3];
226             c_reg[13] += block_a[3][13] * b_reg[3];
227             c_reg[14] += block_a[3][14] * b_reg[3];
228             c_reg[15] += block_a[3][15] * b_reg[3];
229             b_reg[3] = b[3 * n];
230             b += 4*n;
231             c_reg[0] += block_a[0][0] * b_reg[0];
232             c_reg[1] += block_a[0][1] * b_reg[0];
233             c_reg[2] += block_a[0][2] * b_reg[0];
234             c_reg[3] += block_a[0][3] * b_reg[0];
235             c_reg[4] += block_a[0][4] * b_reg[0];
236             c_reg[5] += block_a[0][5] * b_reg[0];
237             c_reg[6] += block_a[0][6] * b_reg[0];
238             c_reg[7] += block_a[0][7] * b_reg[0];
239             c_reg[8] += block_a[0][8] * b_reg[0];
240             c_reg[9] += block_a[0][9] * b_reg[0];
```

```
241             c_reg[10] += block_a[0][10] * b_reg[0];
242             c_reg[11] += block_a[0][11] * b_reg[0];
243             c_reg[12] += block_a[0][12] * b_reg[0];
244             c_reg[13] += block_a[0][13] * b_reg[0];
245             c_reg[14] += block_a[0][14] * b_reg[0];
246             c_reg[15] += block_a[0][15] * b_reg[0];
247             b_reg[0] = b[0 * n];
248             c_reg[0] += block_a[1][0] * b_reg[1];
249             c_reg[1] += block_a[1][1] * b_reg[1];
250             c_reg[2] += block_a[1][2] * b_reg[1];
251             c_reg[3] += block_a[1][3] * b_reg[1];
252             c_reg[4] += block_a[1][4] * b_reg[1];
253             c_reg[5] += block_a[1][5] * b_reg[1];
254             c_reg[6] += block_a[1][6] * b_reg[1];
255             c_reg[7] += block_a[1][7] * b_reg[1];
256             c_reg[8] += block_a[1][8] * b_reg[1];
257             c_reg[9] += block_a[1][9] * b_reg[1];
258             c_reg[10] += block_a[1][10] * b_reg[1];
259             c_reg[11] += block_a[1][11] * b_reg[1];
260             c_reg[12] += block_a[1][12] * b_reg[1];
261             c_reg[13] += block_a[1][13] * b_reg[1];
262             c_reg[14] += block_a[1][14] * b_reg[1];
263             c_reg[15] += block_a[1][15] * b_reg[1];
264             b_reg[1] = b[1 * n];
265             c_reg[0] += block_a[2][0] * b_reg[2];
266             c_reg[1] += block_a[2][1] * b_reg[2];
267             c_reg[2] += block_a[2][2] * b_reg[2];
268             c_reg[3] += block_a[2][3] * b_reg[2];
269             c_reg[4] += block_a[2][4] * b_reg[2];
270             c_reg[5] += block_a[2][5] * b_reg[2];
271             c_reg[6] += block_a[2][6] * b_reg[2];
272             c_reg[7] += block_a[2][7] * b_reg[2];
273             c_reg[8] += block_a[2][8] * b_reg[2];
```

```
274         c_reg[9] += block_a[2][9] * b_reg[2];
275         c_reg[10] += block_a[2][10] * b_reg[2];
276         c_reg[11] += block_a[2][11] * b_reg[2];
277         c_reg[12] += block_a[2][12] * b_reg[2];
278         c_reg[13] += block_a[2][13] * b_reg[2];
279         c_reg[14] += block_a[2][14] * b_reg[2];
280         c_reg[15] += block_a[2][15] * b_reg[2];
281         b_reg[2] = b[2 * n];
282         c_reg[0] += block_a[3][0] * b_reg[3];
283         c_reg[1] += block_a[3][1] * b_reg[3];
284         c_reg[2] += block_a[3][2] * b_reg[3];
285         c_reg[3] += block_a[3][3] * b_reg[3];
286         c_reg[4] += block_a[3][4] * b_reg[3];
287         c_reg[5] += block_a[3][5] * b_reg[3];
288         c_reg[6] += block_a[3][6] * b_reg[3];
289         c_reg[7] += block_a[3][7] * b_reg[3];
290         c_reg[8] += block_a[3][8] * b_reg[3];
291         c_reg[9] += block_a[3][9] * b_reg[3];
292         c_reg[10] += block_a[3][10] * b_reg[3];
293         c_reg[11] += block_a[3][11] * b_reg[3];
294         c_reg[12] += block_a[3][12] * b_reg[3];
295         c_reg[13] += block_a[3][13] * b_reg[3];
296         c_reg[14] += block_a[3][14] * b_reg[3];
297         c_reg[15] += block_a[3][15] * b_reg[3];
298         b_reg[3] = b[3 * n];
299         a += 16;
300         barrier(CLK_LOCAL_MEM_FENCE);
301         } while (a < a_end);
302         c[0] = alpha * c_reg[0] + beta * c[0];
303         c += n;
304         c[0] = alpha * c_reg[1] + beta * c[0];
305         c += n;
306         c[0] = alpha * c_reg[2] + beta * c[0];
```

```
307         c += n;
308         c[0] = alpha * c_reg[3] + beta * c[0];
309         c += n;
310         c[0] = alpha * c_reg[4] + beta * c[0];
311         c += n;
312         c[0] = alpha * c_reg[5] + beta * c[0];
313         c += n;
314         c[0] = alpha * c_reg[6] + beta * c[0];
315         c += n;
316         c[0] = alpha * c_reg[7] + beta * c[0];
317         c += n;
318         c[0] = alpha * c_reg[8] + beta * c[0];
319         c += n;
320         c[0] = alpha * c_reg[9] + beta * c[0];
321         c += n;
322         c[0] = alpha * c_reg[10] + beta * c[0];
323         c += n;
324         c[0] = alpha * c_reg[11] + beta * c[0];
325         c += n;
326         c[0] = alpha * c_reg[12] + beta * c[0];
327         c += n;
328         c[0] = alpha * c_reg[13] + beta * c[0];
329         c += n;
330         c[0] = alpha * c_reg[14] + beta * c[0];
331         c += n;
332         c[0] = alpha * c_reg[15] + beta * c[0];
333         c += n;
334         }
```

Listing A.1: Source code for kernel chosen for AMD GPUs.

```
1           __kernel __attribute__((reqd_work_group_size(8,
                8, 1)))
```

```
2   void gemm(__global float *a, __global float *b,
        __global float *c,
3   int m, int n, int k,
4   float alpha, float beta)
5   {
6   const int bidx = get_group_id(0);
7   const int bidy = get_group_id(1);
8   const int idx = get_local_id(0);
9   const int idy = get_local_id(1);
10  a += (bidy * 8 + idy) * k;
11  b += bidx * 8 + idx;
12  c += (bidy * 8 + idy) * n + bidx * 8 + idx;
13  __global float *a_end = a + k;
14  float a_reg[2];
15  float b_reg[2];
16  float c_reg[1] = {0.0};
17  do {
18  b_reg[0] = b[0 * n];
19  b_reg[1] = b[1 * n];
20  a_reg[0] = a[0 * 1 * k + 0];
21  a_reg[1] = a[0 * 1 * k + 1];
22  c_reg[0] += a_reg[0] * b_reg[0];
23  c_reg[0] += a_reg[1] * b_reg[1];
24  a += 2;
25  b += 2 * n;
26  } while (a < a_end);
27  c[0] = alpha * c_reg[0] + beta * c[0];
28  c += 1 * n;
29  }
```

Listing A.2: Source code for kernel chosen for NVIDIA GPU.