GPU L1 Cache Model for OpenCL Programs

Ewan Crawford
Supervisor: Christophe Dubach

Undergraduate 4th Year Project
Computer Science
School of Informatics
University of Edinburgh

2014

Abstract

Graphics processing units (GPUs) have evolved beyond their original graphical purpose into the field of general purpose computing. This has been facilitated by the development of complex GPUs built with the aim of improving the performance of applications created specifically to run on the GPU, using techniques including a cache hierarchy. Designing efficient programs however is challenging as resources need to be managed between a large number of concurrent threads, requiring developers to better understand how well their software takes advantage of the GPU cache.

To solve this problem we have created a traced based simulation which predicts the cache performance of OpenCL programs. The benefit of our solution is that it provides machine independent cache statistics by using a scheduler to reorder the trace from the user'+s platform. This dissertation details the components of our simulation and how it was validated against GPU hardware to show that we can indeed estimate the L1 cache performance of specific OpenCL kernels.

Table of Contents

Chapter 1

Introduction

Graphics processing units (GPUs) are parallel processors that have undergone major development since their inception in 1999 [17], allowing the scope of their functionality to extend beyond solely graphics and into the realm of parallel applications. These parallel applications are written in high level languages and designed specifically to run on the GPU, an approach called the GPGPU model. Specifically the high level parallel language we are using is OpenCL, which has the advantage of being able to run on any platform and defines functions called kernels to run on the GPU. One of the many ways modern GPUs seek to improve the performance of GPGPU applications is through the use of a cache hierarchy. The benefit of a cache in a GPU is not to directly hide latencies as in a CPU, since multi-threading can accommodate this through thread switching, instead it is to reduce the number of expensive off-chip memory accesses.

1.1 Goal

Our aim is to model the bottom level of the cache hierarchy on a single core, called the L1 cache, so that given an OpenCL [7] kernel we can estimate it's cache performance. This will allow developers without access to a machine with a GPU device to understand how well their kernel utilizes the cache by being able to see the consequences of any changes to the kernel. In addition to helping programmers improve the locality of their programs cache models have also been used to help compilers select the best optimizations and as an aide to cache designers when trying to select the best design configurations.

1.2 Approach

To construct our GPU cache model we have created a trace driven simulation consisting of thread scheduler and L1 cache components which processes traces of memory accesses from real GPGPU program execution. These memory accesses are obtained using the LLVM [9] compiler infrastructure to produce a copy of the original program

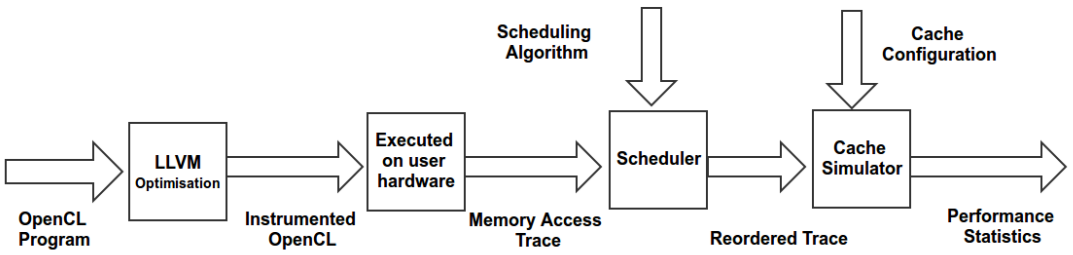


Figure 1.1: Simulation Data Flow

instrumented so that it records memory accesses when they occur. On execution this modified program returns a trace in the sequence executed by the user's hardware, which may be a CPU or GPU. We abstract away the instrumentation process from the user by forcing the OpenCL programs to be coded using a wrapper, which allows us to automate our whole simulation so it is more extensible. After we obtain our trace it is processed with our thread scheduler, and then our L1 cache simulation, which finally calculates relevant statistics that we compare against real hardware for validation. This complete data flow is shown in figure 1.1.

Our trace is gathered using LLVM which features an OpenCL front end that lets us compile our kernel into an intermediate representation (IR) where it can be modified. Two LLVM transformations (chapter 4) are used to instrument the IR, each of which are converted into separate executable kernels. Our first instrumentation is referred to as the preliminary instrumentation and finds the number of memory accesses our trace will contain. While the second instrumentation returns the trace data. After inserting our instructions we use compiler back end Axtor [24] to transform the instrumented IR back into an OpenCL kernel which we can run. The resulting memory access trace from execution is then dumped into a file for input into the thread scheduler.

Since the traces we obtain are machine dependent on the users device the only guaranteed order is with respect to accesses from the same thread. We therefore use an intermediate thread scheduler (chapter 5) to reorder each trace before processing it with our cache simulation. This gives us a new execution order representative of a NVIDIA Fermi [10] GPU where instructions are executed in the SIMT (Single Instruction Multiple Threads) model. Here threads that are executing the same instruction are grouped into batches of 32 called warps and executed concurrently. Our scheduler is not limited to one scheduling policy however it can also be configured with other algorithms like round robin and sequential.

Our cache simulator (chapter 6) then takes the rearranged trace from our scheduler and processes it in a similar way to a single GPU L1 cache, returning statistics such as number of read misses and miss categorization. The L1 cache is focused on as it's the lowest layer of the cache hierarchy and so simplest to recreate directly from the trace. However it does present some problems in terms of machine independence as different GPUs have different numbers of cores, influencing our design choice to simulate just a single cache. Our L1 cache simulation also has to deal with memory coalescing where multiple threads in a warp access memory as a single request.

Our cache model is validated against hardware across three different benchmarks using the setup in chapter 7. All of these benchmarks contain large amounts of data reuse so that the cache is frequently accessed and therefore plays a crucial role in performance. Results in chapter 8 show that we can successfully estimate the miss rate within at most 6% error for all the benchmarks.

1.3 Advantages of Our Solution

Our simulation has the benefits of being lightweight and architecture independent, since although GPU simulators exist [2] they are more monolithic and simulate the complete GPU environment rather than just the cache. As a result they are not as effective for cache analysis due to the lack of configurability and latency introduced from all the extra information simulated. Our solution also has the advantage of being machine independent thanks to the thread scheduler which rearranges the memory accesses, so making our simulation more flexible as the user execution platform doesn't matter.

Trace based simulations are also favoured as the standard method for investigating cache design and performance as opposed to mathematical models. Since for mathematical models it can be hard to derive a good representation of practical programs. Whereas simulations also provide a simpler way to modify the architecture and workload being analysed. However Smith [26] notes some drawbacks to the simulation approach in that traces sometimes only represent a sample of the real workload due to overheads like I/O and the operating system. These factors are less prevalent in a GPU but our trace nevertheless focuses solely only on kernel execution and not any OpenCL setup. In particular we are only concerned with costly off-chip global memory, not taking into account much faster private or local OpenCL memory located on the multi-processors.

Modelling a single cache also allows us to validate our results against the same OpenCL kernel run on physical GPU hardware using NVIDIA's CUDA command line profiler [15] to see the L1 cache performance statistics. This tool provides us with information from one L1 cache, and so by only simulating a single cache we can directly compare our results against the profiler. A lone L1 cache should nevertheless provide a good representation of overall cache performance as it implies that the other L1 caches will have the same performance for applications with symmetrical workloads across cores. Additionally the number L1 cache misses is representative of the number of off-chip memory accesses to slower memory, so any improvements to L1 performance will lead to real program speed-ups regardless of L2 performance.

1.4 Contributions

All the simulation components are written in C or C++ to run in a Linux environment. For experimentation all the benchmark kernels were already available to us, although the OpenCL environment for stencil, section 7.4.3, had to be rewritten so that it used our wrapper.

Contributions:

- `LLVM Transformations` - Creation of two LLVM transformations for instrumenting OpenCL kernels to retrieve a memory trace.
- `Modification of OpenCL Wrapper` - Extending an OpenCL wrapper so that instrumentation is automated.
- `Thread Scheduler` - Creation of a scheduler for rearranging the memory trace to represent Fermi execution.
- `R Graphing Scripts` - Scripts to plot the scheduled memory trace for visualization.
- `Adaptation of Cache Simulator` - Our cache simulator is re-purposed to represent a L1 GPU cache from an existing generic single layer cache simulator written for a previous academic assignment [29].
- `Configuration Scripts` - All these individual components are configured to run together using shell scripts we created that allow us to automate the running of multiple experiments and setup environmental variables.

Chapter 2

Background

2.1 GPU Architecture

In order to create our GPU cache model we need to be familiar with modern GPU architecture and a good product to base our model on is NVIDIA's Fermi architecture [10], NVIDIA's first GPU architecture with a true cache hierarchy. Fermi is based on the previously successful G80 architecture also using the single-instruction multiple-thread (SIMT) execution model where multiple threads are processed simultaneously using a single instruction.

Fermi GPUs are typically made up 16 streaming multiprocessors (SM) which can can each execute one or more OpenCL workgroups. Each SM consists of 32 CUDA cores, figure 2.1, which can execute a floating point or integer instruction per clock cycle for a thread.

Each SM schedules threads in groups of 32 concurrent threads called a warp, using two schedulers and two instruction dispatchers, see figure 2.2. One instruction from each warp is then issued to a group of sixteen cores, sixteen load/store units, or four Special Function Units (SFU). Therefore a warp is executed over at least two clocks cycles, 16 threads are processed and then the other 16 threads in that warp are processed.

In terms of memory there is a L1 cache per SM and a single shared L2 cache accessible by all threads in addition to high latency global memory. The SM on-chip memory can be configured as either 48 KB of shared memory with a 16 KB L1 cache or as 16 KB shared memory with 48 KB of L1 cache. Shared memory enables threads in the same workgroup to cooperate and in OpenCL would be in the local memory address space. To allow the L1 caches to be as fast as possible they are not coherent and have a write-through policy, with 4-way associativity for the 16 KB configuration and 6-way for the 48 KB.

Figure 2.1: Fermi Streaming Multiprocessor (SM). Source: [10]

All 16 SMs also share the 768 KB L2 cache which services all loads and stores to global memory. Both cache levels have a 128 byte line size, enough for 32 integer elements. Notice that the size of a warp is 32 threads, so it is possible for every thread in a warp accessing consecutive data items from a single instruction to hit the same cache line. Accesses from threads in a warp are coalesced into a single access if they hit the same cache line, called perfect coalescing if every thread in the warp hits the same line.

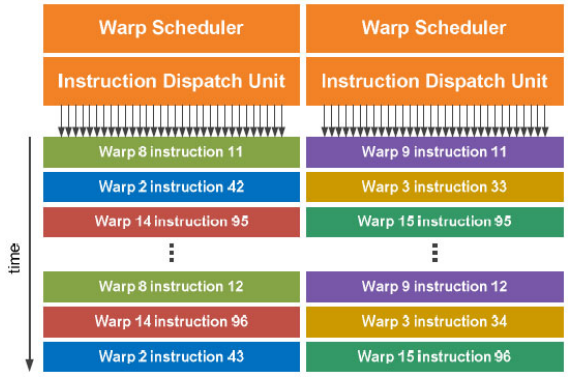


Figure 2.2: Fermi Warp Scheduling. Source: [10]

The L2 write-policy is not documented but understood with reasonable certainty [1] to be write-back on a hit and write-allocate on a miss. The replacement policy for both levels is less well understood but assumed to be a LRU variant [14].

2.2 Caches

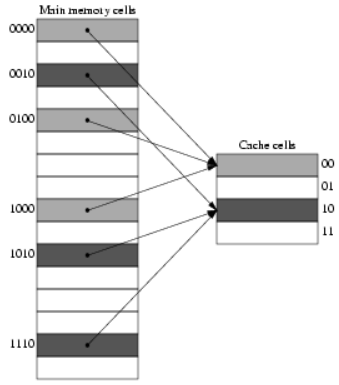


Figure 2.3: Cache Addressing.[Source:"http://linuxgazette.net/102/pramode.html"]

Caches have been designed to hide the latency of memory operations by caching the most frequently accessed data close to the processor. These caches are relatively small but still effective as they take advantage of the temporal and spatial locality prevalent in programs. Temporal locality is where if a particular memory address is referenced, then it is likely referenced again in the near future, such as in a loop. Spatial locality however is where a specific memory address is referenced, then it is likely that nearby memory locations will be accessed shortly, like array items.

All caches are organized into sets, which consist of lines, with the number of lines per set defined as the associativity. Each cache line contains data from consecutive memory locations, capitalizing on spatial locality, and is identified by a tag. This tag is usually the highest order bits of the desired memory address, which is compared against the tags of all the lines in the relevant set. As a result of the latency in searching all these tags a cache designer must weigh up the benefits of increased associativity, which reduces conflict misses, against the latency of searching all the lines.

When a program needs to access data it first consults the cache to see if there is a matching entry, if so this is a cache hit, otherwise it is a cache miss. Misses are expensive as it results in the cache having to fetch the data from larger, slower memory and can arise from three situations categorized as conflict misses, capacity misses, and cold misses. Cold misses are those which occur because they are the first reference to a location in memory, so without prefetching there is no way to prevent the miss. Capacity misses however arise because of the finite size of the cache, where a line must be evicted to make space for the new reference. The way this evicted line is chosen is called the replacement policy and common strategies include least recently used (LRU)

and least frequently used (LFU), although LRU is the most common as it is computationally cheapest to calculate. Finally conflict misses are those misses which could have been prevented if an entry had not been evicted previously.

Caches are not just accessed for reads however they also need to be concerned with writes to ensure cached data is synchronized with memory, the method used to deal with writes is called the write policy. Common write policies are write-through, where data is written synchronously to both the cache and backing store, and write-back where data is written only to the cache. Write-back is more complex as the write to backing store is delayed until the line is evicted, requiring blocks to be marked as 'dirty' if they have been modified. There are also two approaches to the case of a write miss, one being write allocate used with a write-back policy, where the write miss data is loaded into the cache then is followed by a write hit. Write-through caches on the other hand typically use no-write allocate which writes data directly to backing store.

2.3 OpenCL

GPGPU programming allows applications to be written specifically for the purpose of being executed on a GPU. This is made possible through libraries developed to let users utilize the GPU using programming languages with which they are familiar. NVIDIA provides a popular framework called CUDA [21] which only works on NVIDIA platforms and is the dominant propriety language. However OpenCL [7] is the leading open source framework which utilizes similar concepts and is what we have chosen to use as it's compatible with more platforms and we have tools available for it.

OpenCL programs can run on a variety of platforms and lets the user define functions called 'kernels' to be executed by the GPU. The environment in which kernels are run is specified by OpenCL APIs which are written in C++ and consists of six main data structures: device, kernel, program, buffer, command queue, and context. The relationship between these structures is shown in figure 2.4

- `Device` Represents the piece of hardware which will be executing the kernels and can be either a CPU or GPU.
- `Kernel` Functions which are to be executed on the device, written in a syntax based on C99.
- `Program` Used to build and compile kernels from either source code or binary.
- `Buffer` Array used to share data between devices, addressed by the kernel through a pointer parameter.
- `Command Queue` Each device receives kernels through the command queue.
- `Context` Allows devices to receive kernels and transfer data.

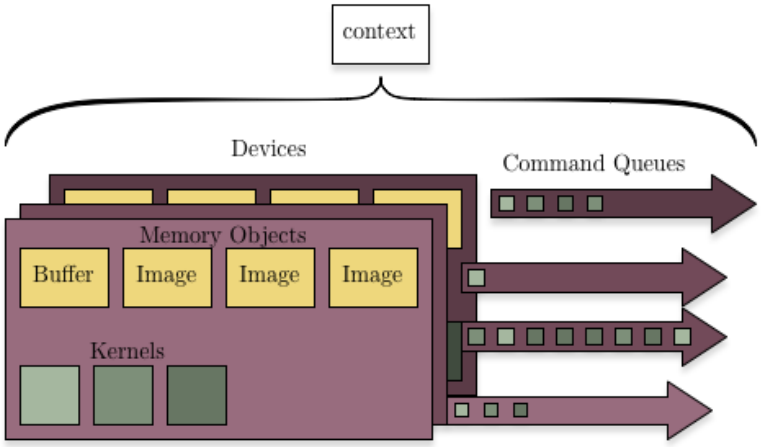


Figure 2.4: OpenCL data structure relationships.[Source:"http://mygsoc.blogspot.co.uk/"]

In the OpenCL execution model kernels are executed by one or more workitems, which are individual threads of executions. Each workitem is assigned to a workgroup, with each workgroup being executed on a compute unit (multi-processor) in hardware. The user can create any number of workitems and workgroups however if the chosen device has M compute units, and if the program has N workitems per workgroup, then only MN workitems can be executed by the kernel at any one time.

Workitems are identified by dimensional indices both globally amongst all threads and locally for their workgroup. Assignment of workitems to workgroups is based on global id and shown in figure 2.5. Although the number of dimensions is not bounded by OpenCL the practical limit on hardware is 3. The set of active threads is all those workitems which are mapped to a specific SM and can be made up of one or more workgroups assigned to each SM with no predefined ordering.

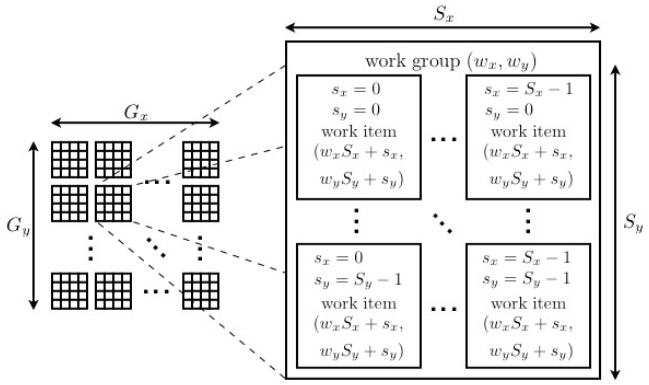


Figure 2.5: OpenCL workgroup structure. Source: [3]

OpenCL device memory is categorized into four address spaces:

1. Global memory which stores data for the entire device and can be used by all workitems but is relatively slow to access.
2. Constant memory which is read-only global memory.
3. Local memory that is shared by all workitems in a workgroup but isn't as large as global or constant memory.
4. Private memory which stores data for an individual workitem.

Memory barriers or fences are points in execution defined the kernel which no thread can pass until all the other workitems have reached that barrier, useful for synchronization. Barriers can't be conditional and if they are in a loop then all the threads must see the barrier every iteration to prevent deadlocks. We need to be aware of these when scheduling our trace, since accesses can't be moved across them.

Chapter 3

Literature Review

In this chapter previous research in the field of GPU caching and scheduling is presented. Beginning with evaluation of similar work on L1 Fermi cache modelling before going on to describe proposed methods to improve GPU performance through smarter scheduling.

3.1 Cache Modelling

3.1.1 Related Work

A lot of existing research has been carried out in the field of GPU optimization, the most relevant of which to our work is Nugteren et al's proposed GPU cache model [14] based on reuse distance theory. Beyls and D'Hollander [8] initially defined reuse distance theory, allowing us to obtain cache hit ratios for sequential execution using a LRU fully associative CPU cache. This is done by using a stack to hold previously accessed cache lines, with the depth of each cache line from the top of the stack defined as the reuse distance. Calculating the miss rates of solely fully associative caches can still be useful for analysis though, as shown in work by Smith [25] which proves that the miss rates for set associative caches can be derived from their fully associative counterparts.

Nugteren et al's work extends this reuse distance theory to GPUs parallel execution model by generating a memory access trace using round robin scheduling between warps in the set of active threads for a specific core. Although to account for warp divergence round robin scheduling is intentionally made not fair by using a FIFO queue to contain warps. Allowing each warp to be delayed for a proportional time after execution to account for it's latency until being placed back in the queue.

Memory latencies for each access are also introduced to the extended reuse distance theory as latency is more of a influence for GPUs than CPUs. To achieve this incrementing timestamps are introduced for each instruction in the trace, with a latency until each access takes effect in the cache. Latency misses are therefore highlighted as

a possibility, where requests miss because a previous access to the requested cache line is still in flight. Another extension by Nugteren et al was to use a single reuse stack for each set, taking into account the associativity in GPU caches because searching through tags adds to latency.

Nugteren et al weren't the first people to try to model GPU caches based on reuse distance theory however, Tang et al [28] also created an earlier model for analysing GPU cache misses using some simplifications to create a stack distance profile for a single workgroup. A single workgroup was used based on the assumption that there is only a small chance adjacent workgroups will be assigned to the same core and so generally there will be no inter-workgroup reuse. The technique Tang et al use to create a stack distance profile for parallel execution is quite different from Nugteren et al, as their technique involves using a kernel access vector proposed by Xue at al [31] to define relationships between accesses at the instruction, workgroup, and warp level.

Although Tang et al produced good experimental results their experimental setup is limited in terms of validation since it's results are only compared against a simulator, which does not have the same credibility as validation against real hardware. Additionally the set of kernels validated against is limited to those which can be statically analysed since the model doesn't have any capabilities to create a trace and reconstruct parallel execution.

3.1.2 Relevance to Our Work

Nugteren et al's paper is relevant to our work in that it shares similar aims in modelling a GPU L1 cache but also because of commonality in it's implementation. Both our models use a trace obtained from kernel execution with no predefined global ordering and attempt to recreate GPU scheduling. However Nugteren et al use a GPU emulator complete with tracer to produce a list of memory accesses whereas we trace real kernel execution.

Both our models are also validated using NVIDIA's command line profiler [15], although Nugteren et al's work uses both L1 cache configurations when we only validate against the smaller 16KB configuration. Our stencil benchmark (section 7.4.3) is also taken from the Parboil [27] benchmark suite used by Nugteren et al, who also experiment with the PolyBench/GPU [23] suite.

The main difference in methodologies however is that whereas we process our scheduled trace using a cache simulation Nugteren et al perform coalescing beforehand and then apply their extended reuse distance theory to calculate the miss rate. Tang et al's research into the mapping between workgroups and cores is also useful to us when trying to choose workgroups to assign to our cache simulation in section 6.4.

3.2 Scheduling

3.2.1 Related Work

GPU scheduling policies have also been the subject of much research with Narasiman et al [13] highlighting some shortcomings of existing GPU scheduling policies in their paper 'Improving GPU performance via Large Warps and Two-Level Warp Scheduling'. The authors believe that GPU cores are underutilized despite warp scheduling, as thread divergence in a warp from conditional branches results in a core having to execute each divergent path sequentially with fewer threads than optimal. The other deficiency in warp scheduling outlined by Narasiman et al is that GPU cores don't mask long latency operations optimally due to round robin scheduling between warps. Arguing that all warps can reach the same long latency instruction at the same time, with no idle warps that can be scheduled to hide the latency. However allowing warps to advance at differing rates would impact locality as a cache line used by one warp is likely to be accessed again by other warps.

The problem of thread divergence in warps has been raised before by Fung et al [5], with the proposed solution being dynamic warp formation (DWF), where the diverged threads of several warps branching to the same target could be grouped together to compose a new complete warp. This work has since been superseded in a paper by the same authors [4] who raise the point that DWF requires a large number of warps to progress at roughly the same pace. Therefore a technique called thread block compaction is proposed, where at a divergent branch threads are compacted and formed into a new warp. These new warps then execute until they reach a reconvergence point where they synchronise again and the threads are compacted into their original arrangements before divergence. This thread block compaction strategy ensures that as many threads as possible sleep at a branch or reconvergence point, allowing other threads to be optimally scheduled.

Based on this work Narasiman et al [13] propose two techniques for improving scheduling, first of all using fewer but larger warps in a large warp formation (LWF) to better facilitate DWF. Secondly implementing two-level round robin scheduling between warps to reduce the impact of long latencies while still maintaining associativity. In this LWF we can create SIMT width sized sub-warps which will be formed dynamically from the active threads in a larger warp, even in the presence of branch divergence. After a large warp has reached a divergent branch it is not considered again for scheduling until all its sub-warps are complete, because otherwise it is unknown whether the larger warp has diverged or not. Choosing the size of the large warp however can be complex as although larger warp sizes give more potential for efficient sub-warps, if the majority of threads have reached a convergence point then they will have to stall while waiting for a few threads to catch up.

The proposed two-level warp scheduling policy works by grouping concurrently executing warps into fixed sized fetch groups, with each fetch group assigned a priority. All the warps in the same fetch group have equal priority and are scheduled round robin amongst each other, until they are all stalled on a long latency operation. At this

point the fetched group is switched to the group with the next highest priority and it's own priority set to the least, so fetch groups are also scheduled round robin. Like LWF the size parameter must be set carefully, as to be efficient the fetch size must be greater than the number of pipeline stages. However larger fetch groups take longer to process and reach a stalling point, also resulting in a greater number of warps stalling at the same time with fewer left to hide the latency.

3.2.2 Relevance to Our Work

These possible scheduling strategies are of interest to us as it would be possible to implement both LWF and two-level warp scheduling with our scheduler as future work. This would allow us to evaluate their use of the cache and see they would indeed achieve the claimed performance improvement.

3.3 Summary

In this chapter we have seen how Nugteren et al have used a similar technique as us to model Fermi L1 caches by extending reuse distance theory to predict cache miss rates on an emulated memory trace. Nugteren et al also rearrange this trace to represent GPU execution but perform coalescing at the same time, whereas this is integrated into our cache simulation.

The scheduling section outlines large warp formation and two-level warp scheduling strategies for improving GPU performance. Both of which could be implemented in our scheduler to see if their utilization of the cache supports these claims.

Chapter 4

LLVM Instrumentation

Before we can do anything with our trace based simulation we need to be able to obtain traces of working programs so that we have something to process. In this chapter we detail how this is done using the LLVM compiler infrastructure to add instructions to the compiler's intermediate representation of kernels. Section 4.1 gives an overview to the whole workflow used to obtain a trace. With sections 4.2 and 4.3 describing the two separate instrumentations that need to be performed. The technique used in the second instrumentation so that we can later reconstruct an ordering between threads is described in detail in section 4.4.

4.1 Overview

Let us first of all define the information that needs to be contained in our trace so that we can later group accesses from different threads together to reconstruct the SIMT model.

- `Memory Address` Location in memory accessed, section 4.3.1.
- `Read/Write` Read or Write to memory, section 4.3.1.
- `Instruction Name` Instruction the access was issued by, section 4.3.1.
- `Thread ID` Global ID of the workitem which made the access, section 4.3.2.
- `Loop Iterations` Iteration of any loops the instruction was inside when executed, section 4.4.

All of this information can be obtained by using compiler infrastructure LLVM to instrument the kernel so this data can be returned. LLVM has an OpenCL frontend that lets us transform the kernel into an intermediate representation (IR) where it can be modified with the LLVM optimizer tool [12]. It is then through our own transformations written using the C++ API for the optimizer that we can add the instructions to obtain our trace. LLVM backend Axtor [24] is then used to change the instrumented IR back into OpenCL which we execute. This complete process is shown in figure 4.1.

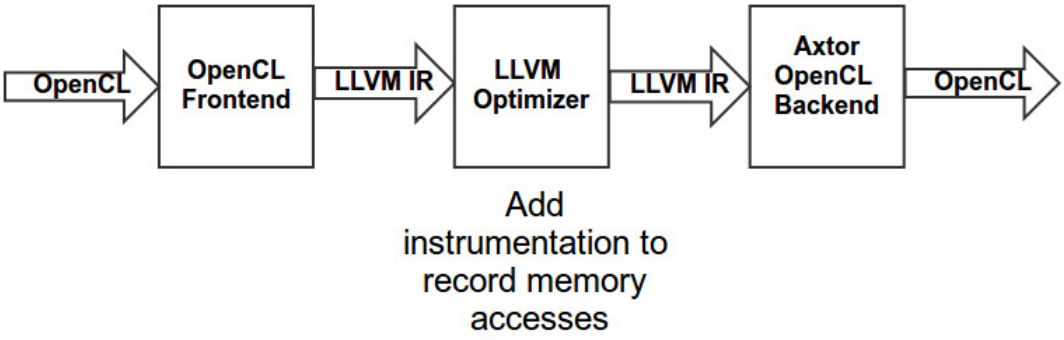


Figure 4.1: LLVM instrumentation steps.

Modifying every OpenCL program we want to trace manually however is cumbersome and as a result we utilize an OpenCL wrapper developed by Informatics PhD student Alberto Magni. This wrapper encapsulates central OpenCL objects so that they are abstracted and easier to work with. By modifying this wrapper so it also carries out the instrumentation we can ensure that all the parameters of a user's program are compatible with our LLVM transformation without the user having to change their program in any way. However traces can be millions of entries long and in order for our wrapper to allocate the correct amount of space for our trace we need to know it's length a priori. Since we can't do this statically we use two LLVM transformations, the second returns all the trace data, while the first preliminary transformation simply finds the length of our trace and checks for any potential problems that might hinder the second pass.

4.2 Preliminary Instrumentation

The primary purpose of this first transformation is to add instructions to the kernel that count the number of global memory accesses. However kernel files can contain many kernel functions and as a result we need to pass in as a parameter the name of the kernel which we want to trace. Once we have found the target kernel our transformation creates a counter shared between all threads by adding a 32 bit unsigned global buffer. The first entry of this buffer stores the counter which is initialized to zero by the wrapper before execution.

The transformation then goes on to examine all the kernel instructions in their static order to see if they are relevant. If the instruction is a load or a store, then only if it is to global memory do we add an instruction to increment the counter. To prevent race conditions each thread increments the counter using the OpenCL atomic increment operation atomic_inc(). This instruction simultaneously reads and increments a value, preventing other threads from reading or writing to that location until the operation completes.

Memory fences and barriers in the kernel also need to be included in our final memory trace so that memory accesses are not scheduled across them. These functions represent points in execution whereupon every thread must commit to memory any loads

and stores preceding the fence/barrier, before any loads and stores made afterwards can be issued. In our preliminary transformation if an instruction is a function call to either barrier() or mem_fence(), then an atomic_inc() instruction is added afterwards to increment the counter.

The other purpose of this preliminary instrumentation is to check for potential issues with the kernel that could affect the second transformation's results. Any issues are signalled to the wrapper by writing to subsequent entries in our added buffer, which the wrapper can then alert the user to. Most of these issues are to do with loop restrictions described in section 4.4.4, however we also need to check for helper function calls that pass a global memory pointer as a parameter. Since if this pointer is referenced in the helper function we cannot properly record all the information about the memory access.

4.3 Trace Instrumentation

Our second instrumentation is carried out to then retrieve all the information defined in section 4.1. To do this we need to add 3 additional 64 bit buffers to our kernel all initialized to the length found in the preliminary trace.

The first location in these added buffers also acts as a counter that is incremented after each memory access with atomic_inc(). We use this counter to point to the next location in the buffers to write the memory access data to, avoiding race conditions that could overwrite entries. We don't need all three buffers to store this counter in their first entries however, so it's stored in the first buffer and the other two remain unused.

4.3.1 Address Buffer

The first buffer we add to the kernel, figure 4.2 , records the address referenced by each global memory access in the last 32 bits. While the first 32 bits store the instruction each access originated from, and whether it was a read or a write.

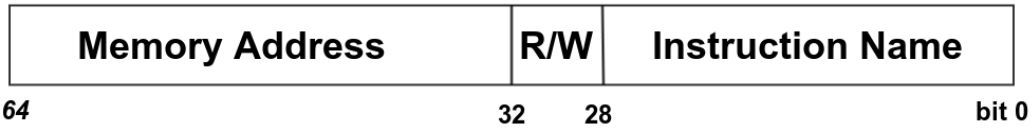


Figure 4.2: First instrumented buffer storing: memory address, read/write, and Instruction Name.

In both read and write memory instructions we can see the referenced 32 bit memory address used as an argument. The address data is therefore easily obtained by adding instructions to bit shift the same address by 32 and store it in our buffer. Whether the

instruction is a read or a write can also be easily ascertained by the type of instruction. Reads are recorded by adding an instruction afterwards writing 0xF to bits 28-32, while writes are denoted by storing the value 0xA.

Uniquely identifying the instruction itself is slightly harder since we need a consistent naming policy for both loads and stores that gives us a static ordering, but can still be returned in 28 bits. Therefore every load and store instruction in the kernel is renamed as 'Mx', where x is an increasing integer for every load or store seen. A base 36 radix is then used to encode the instruction name as an integer so it can fit in the buffer segment and be easily decoded by the scheduler.

4.3.2 Thread ID Buffer

Our next buffer instrumented in figure 4.3 is used to identify the thread which made the memory access.

Figure 4.3: Second instrumented buffer used for global id in 3 dimensions.

In OpenCL a thread has an id in multiple dimensions which can be retrieved using the 'get_global_id(int x)' function to get the id of the thread in dimension x, returning 0 if that dimension is undefined. The practical limit to the number of dimensions on hardware however is three. We therefore assign 20 bit segments in our buffer for each dimension id. To get each value we add a get_global_id() call after the memory access and bit shift the result to the according buffer segment.

When recording memory fences and barriers we still write the id of the thread which encountered the function to this second buffer as before. However the first buffer has solely either integer 1 or 2 written to it depending on whether the synchronization is local or global respectively, although we deal with both in the same way. The final buffer discussed next is unneeded since synchronization points can't be inside conditionals.

4.4 Loop Buffer

Our final buffer is used to store a snapshot of any loops a memory access was inside when it executed. This information is needed so that we can recreate hardware SIMT execution in our scheduler by grouping accesses with the same loop snapshots together.

4.4.1 Purpose

Fermi hardware executes instruction in a SIMT pattern, where a 32 thread warp makes a single concurrent access. In order for our simulator to be able to reconstruct these concurrent accesses from a potentially sequential trace every access needs a timestamp. All the accesses to the same instruction, with the same timestamp, can then be grouped together.

Our initial approach was to give each global memory instruction a shared integer timestamp, which would increase every time the instruction was accessed by a thread. The idea being that any accesses from the same instruction within a timestamp boundary of around 32 could be grouped together. However this idea was flawed in that it was not machine independent, since threads running sequentially could increase the timestamp rapidly in a loop. So it is unknown how accesses made by later threads to the same instruction should be grouped.

Instead to be machine independent there needs to be a private timestamp for each thread associated with every global memory access. This allows us to compare private timestamps across threads so that accesses can be matched up. Since there is no recursion in OpenCL because of GPU hardware limitations the only way an instruction can be called more than once in a kernel function is if it is inside a loop. Therefore we implemented a solution where a private timestamp tells us exactly what loop iteration the instruction was executed on. Instructions which should be combined together are therefore those which have the same loop iterations and are executed by threads in the same warp. The third and final extra 64 bit buffer our instrumentation adds to the kernel therefore acts as a loop timestamp for each access.

4.4.2 Buffer Structure

Figure 4.4: Third instrumented buffer, used to take a snapshot of any loops the memory instruction was inside when executed.

Figure 4.4 shows how the final added buffer is divided up into sections of 20 bits which each hold information about a single loop. We therefore only have enough space for loops to be nested three deep. Each section is partitioned into space for a 16 bit loop iteration and 4 bit loop id.

The loop id identifies which loop the associated iteration is for and is assigned to each loop statically in ascending order at the start of the LLVM transformation. Due to this ordering we can infer the nesting of loops from the id because no loop can be nested inside a loop with a larger id. The loop iteration value stores the number times the loop

has iterated when that instruction was executed and is reset to zero on loop termination to make processing nested loops easier.

4.4.3 Recording Loop Iteration

Before examining the kernel instructions for any memory accesses the first thing our trace transformation needs to do is find all the kernel loops and assign each an id. We achieve this by examining all the basic blocks in the kernel, since in LLVM IR the naming of basic blocks follows a specific pattern for loops. Basic blocks are defined as sections of code which only have one point of entry and exit. All loop bodies have basic blocks named so that they are appended with the suffix '.body', while the exit basic blocks called on loop termination have suffix '.end'. We need to record the relationship between body blocks and exit blocks since loops have their iteration variables reset to zero on termination. Therefore after we have discovered a loop body to find it's exit block we examine it's predecessor basic block, since the predecessor will have a branch condition to the exit block.

We then need to create a private variable for each loop at the beginning of the kernel so that each thread can store the loop iterations of all the loops. So that these variables are updated on each iteration we add an instruction in each loop body basic block to increment it's variable. Each loop exit basic block also has an instruction added to reset it's loop iteration variable to zero.

Now that the loop iterations will be recorded our transformation examines each instruction in the kernel to look for global memory accesses. When a global load or store is seen then only loops which currently have non zero iteration values are recorded, as these are the loops the instruction is currently inside. By using a comparison instruction to compare loop iterations with zero we can create a mask that is used to hide irrelevant loops, before logically ORing each loop with the existing buffer. Each active loop's data is shifted to it's 20 bit segment in the buffer using an offset which increases when it sees a loop not masked to zero.

4.4.4 Constraints

Due to the limitations of a 64 bit buffer there are some constraints involving the kernels we can accurately trace. As a result in addition to finding the length of the trace our preliminary instrumentation also needs to check for loop situations which would break our implementation of recording loop data.

Therefore our preliminary trace too identifies all the loops and counts their iterations by creating private variables which increment using the loop body and exit basic blocks. However after finding all the loops our first transformation then checks that there are less than 2^4 loops in the kernel. Since our loop id assigned space is only 4 bits per loop. If this condition is violated 0x15 is written back to the second element in the only instrumented buffer to signify that this violation has occurred.

Additionally because 16 bits are assigned to the loop iteration there can never be more than 2^{16} loop iterations. A greater than comparison instruction is therefore added to each loop body to check this condition with the result ORed then written back to element five in our added buffer.

Finally since we only have space for three loops in the buffer we need to check that no global memory accesses are nested more than three loops deep. Thus in every loop body we instrument a comparison instruction to see if the number of loop iteration counters above zero us greater than three. This result is then ORed with and stored back to position four in the instrumented buffer.

4.5 Wrapper

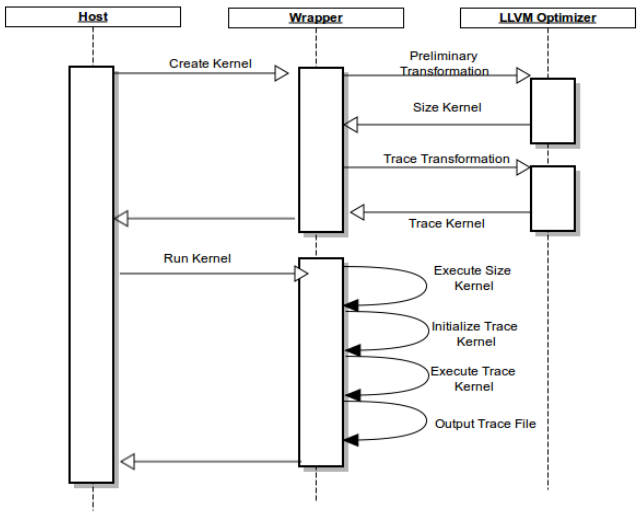


Figure 4.5: Wrapper Data Flow

Before we can execute these instrumented kernels we first need the wrapper to generate them from the original kernel. Figure 4.5 shows the data flow allowing the wrapper to instrument and execute the kernel we want to trace.

This is achieved through modifications to the Program and Kernel wrapper abstractions of the cl_program & cl_kernel OpenCL objects. Our modified versions of Program and Kernel contain two of each encapsulated OpenCL object, one for the results of each transformation. To create these new objects we first read the original kernel file into a string and find the position of the kernel function we want to trace. Next we add the parameters for our extra buffers to the kernel source by simply inserting them into the string at the correct place. To then run the LLVM transformations to obtain the instrumented programs we call a script for each instrumentation which runs the optimizer tool. The script then calls the Axtor backend to convert the optimized IR back into OpenCL which we can read from a file and use to create our cl_program objects.

Finally the last wrapper object we need to change is Queue which is an abstraction of the OpenCL object for executing kernels, cl_command_queue. Here when the Kernel object is run we perform the following steps:

1. Kernel from preliminary instrumentation is run to give us trace length.
2. Check for any warnings and initialize kernel from trace transformation with returned trace length.
3. Execute kernel from trace instrumentation.
4. Dump instrumented buffers into an output file. Where each file line is a concatenation of each of the buffer elements at that position separated by a pipe symbol '|'.

4.6 Example

```
__kernel void reset(__global int* data){

   if(get_local_id(0) == 0) {
      data[get_global_id(0)] = 0;
   }
}
```

Listing 4.1: Simple kernel function setting data elements to zero at positions corresponding to the threads which are the first in their workgroup

To illustrate the trace instrumentation process we have an example kernel in listing 4.1 using one dimension. This kernel simply sets the data buffer to zero at the global index of any thread which is the first in it's workgroup. Therefore there is at most a single write to global memory for each thread.

Before we can pass this kernel into our trace transformation however we need to add the three extra buffers required to return the data. Listing 4.2 shows the addition of these buffers.

```
__kernel void reset(__global long* trace, __global long* ids, global long* loop_ctr,
    __global int* data){

   if(get_local_id(0) == 0) {
      data[get_global_id(0)] = 0;
   }
}
```

Listing 4.2: Kernel function with buffer parameters added to return trace data

When Axtor is used to convert this instrumented IR back into OpenCL in listing 4.3 the extra instructions have a clear impact on the size of the kernel because LLVM IR uses temporary variables. This impacts run time by making the kernel slower to execute but does not affect the correctness of the trace which is what we are concerned with.

We can see the original store instruction near the bottom on line 49, with lines 29-36 containing the instructions for the first address buffer. Where atomic_inc() on line 31

increments the counter pointing to the position in the buffers being written to. Lines 37-47 features instructions for the thread id buffer as we can see the calls to get_global_id() with different parameters. As there are no loops in this example line 48 writes zero to the loop buffer

```
1  __kernel void reset(__global long* trace, __global long* ids, __global long* loop_ctr
       , __global int* data)
2  {
3     int reset_0_1;
4     bool reset_0_2;
5     long reset_1_20;
6     int reset_1_0;
7     int __global* reset_1_3;
8     int reset_1_4;
9     long reset_1_5;
10    long reset_1_6;
11    long reset_1_7;
12    long reset_1_8;
13    int reset_1_11;
14    long reset_1_12;
15    int reset_1_13;
16    long reset_1_14;
17    long reset_1_15;
18    long reset_1_16;
19    int reset_1_17;
20    long reset_1_18;
21    long reset_1_19;
22    int reset_0_0;
23
24    reset_0_0 = (int)(0x00000000);
25    reset_0_1 = get_local_id((int)(0x00000000));
26    reset_0_2 = (reset_0_1==(int)(0x00000000));
27    if (reset_0_2)
28    {
29       reset_1_0 = get_global_id((int)(0x00000000));
30       reset_1_3 = (__global int*)(&(trace[0]));
31       reset_1_4 = atomic_inc(reset_1_3);
32       reset_1_5 = convert_long(&(data[reset_1_0]));
33       reset_1_6 = (reset_1_5<<(long)(0x0000000000000020));
34       reset_1_7 = (reset_1_6|(long)(0x00000000A0000000));
35       reset_1_8 = (reset_1_7|(long)(0x0000000000000319));
36       trace[reset_1_4] = reset_1_8;
37       reset_1_11 = get_global_id((int)(0x00000000));
38       reset_1_12 = as_long(convert_ulong(as_uint(reset_1_11)));
39       reset_1_13 = get_global_id((int)(0x00000001));
40       reset_1_14 = as_long(convert_ulong(as_uint(reset_1_13)));
41       reset_1_15 = (reset_1_14<<(long)(0x0000000000000014));
42       reset_1_16 = (reset_1_15|reset_1_12);
43       reset_1_17 = get_global_id((int)(0x00000002));
44       reset_1_18 = as_long(convert_ulong(as_uint(reset_1_17)));
45       reset_1_19 = (reset_1_18<<(long)(0x0000000000000028));
46       reset_1_20 = (reset_1_19|reset_1_16);
47       ids[reset_1_4] = reset_1_20;
48       loop_ctr[reset_1_4] = (long)(0x0000000000000000);
49       data[reset_1_0] = (int)(0x00000000);
50       return;
51    }
52    else
53    {
54       return;
55    }
56 }
```

Listing 4.3: Instrumented Kernel IR from 4.2 converted back into OpenCL using Axtor which will be executed through the wrapper

Chapter 5

Thread Scheduling

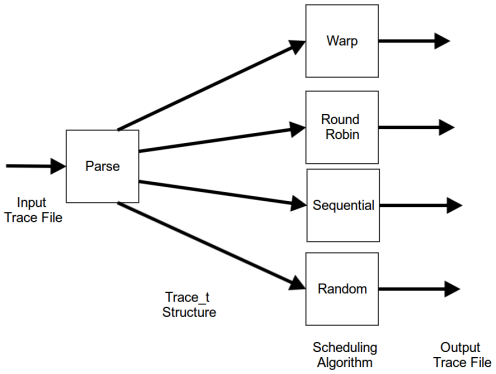


Figure 5.1: Scheduler data flow.

Now that we have obtained a trace of kernel execution using the technique from chapter 4 we need to rearrange the trace before processing it with our cache simulator. This is because the kernel could have been run on any platform but we want to simulate the cache performance specifically for Fermi GPU execution. The trace therefore needs to be reordered to replicate this without assuming any global ordering between threads.

In this chapter we begin with section 5.1, describing how the memory trace is read into a data structure to better facilitate rearrangement. Section 5.2 specifies some of the naive scheduling policies implemented to visualize algorithmic differences using R [30] graphing software. How we recreate Fermi scheduling is detailed in section 5.3, where threads are grouped into concurrently executing warps.

5.1 Parsing

Before we can begin to schedule the trace we need to understand it and make the data accessible. This is done by converting the trace into data structures which are more easily rearranged.

Listing 5.1 shows a small example memory trace file that would be used as input to the scheduler. Here we can we can see how each of the instrumented buffer values are separated with a '|' character and the first line contains workgroup size metadata. The first eight entries show alternating reads from two threads. While the last entry 0x1|0x10|0x0 is a local memory fence call from thread 16 since it contains only the integer one in the first buffer. Consecutive hyphens mark end of execution, needed because there is the possibility of kernels performing multiple iterations and so we have to be careful not to rearrange accesses across iterations.

```
local size:16 16 1
0x9CAE000F0000319|0x0|0x10000000000
0x9CAE000F0000319|0x10|0x10000000000
0x9CBB000F000031A|0x0|0x10000000000
0x9CBB040F000031A|0x10|0x10000000000
0x9CAE004F0000319|0x0|0x20000000000
0x9CAE004F0000319|0x10|0x20000000000
0x9CBB1C0F000031A|0x0|0x20000000000
0x9CBB200F000031A|0x10|0x20000000000
0x1|0x10|0x0
----------------
```

Listing 5.1: Memory Trace example scheduler input of two threads 1 & 16 looping over and reading two data structures. Last line 0x1|0x10|0x0 represents a memory barrier call from thread 16 since it's first value is 1.

5.1.1 Structures

We convert the trace into a hierarchy of structures shown in listing 5.2. The trace_t structure holds information regarding the OpenCL environment as well as a pointer to all the individual access structures. Each of these access trace_entry_t structures holds all the data from a memory access or barrier call that is parsed from a single trace line. Since thread id, as well as global and local size, can have up to three dimensions they are best represented using the t_dim_t structure with a value for each dimension. Loop data is also stored in a special structure, loop_t, containing integer arrays for loop id and iteration, where values at the same index are associated with each other. The local_num_threads field of the trace_t structure is populated from the first line of the input file which contains the local size of the workgroups in each dimension.

```
//Thread id struct
typedef struct  t_dim_s
{
        uint32_t D1;      //First dimension id
        uint32_t D2;      //Second dimsension id
        uint32_t D3;      //Third dimension id
}  t_dim_t;

//represents loop data
typedef struct loop_s
{
  int lp_size;              //Number of nested loops in struct
  int labels[3];            //Loop label for up to 3 nested loops
  uint32_t counters[3];     //Loop execution counter
} loop_t;

//Stucture holding data referring to an inidividual entry
typedef struct trace_entry_s
{
    uint32_t mem_addr;     //memory address
    bool read;             //true if read, false if write
    char* inst;            //instruction name
    t_dim_t t_id;          //thread id
    uint64_t index;        //index in trace
    char meta;             //metadata: 'L' or 'G' memory barriers,
                           //          'E' end of trace
    loop_t loop_data;      //loop data
} trace_entry_t;

typedef struct trace_s
{
    trace_entry_t **trace_entries; //trace entries
    int dim;                       //number of dimesions
    t_dim_t num_threads;           //number of threads globally
    uint64_t size;                 //length of trace
    t_dim_t local_num_threads;     //threads in a workgroup
} trace_t;
```

Listing 5.2: Scheduler Structures

5.1.2 Individual Input Lines

After getting the local size the rest of input file is then read into a buffer and parsed line by line into trace_entry_t structures that can be rearranged. For each line we initially find the location of the first split token '|'. If this token is after only one character then we know that this entry represents a memory barrier. In which case the meta field of trace_entry_t is set to 'L' for local or 'G' for global. Depending on if the character before the split token is 1 or 2 respectively. In the case that no split token exists then we check for consecutive hyphens, and accordingly set meta to 'E' for end of execution. Otherwise the entry is a memory access and meta is set to 'N' for not metadata.

If the memory address has dropped a redundant 0 from the front of it's address then we add it back on to make all the lengths uniform, before reading the initial eight hexadecimal characters into the mem_address field. The ninth character can then be classified as 'F' or 'A' to identify the access as a read or write respectively, setting the read trace_entry_t boolean. All the remaining characters before the split are then decrypted using a base 36 radix to get the plain text instruction name that can be stored in string field inst.

The thread id of each trace_entry_t structure can be obtained by converting the string between the split tokens from base 16 characters into an integer and bit shifting. A technique also used for loop data. However we need to keep an updating maximum thread id in each dimension so that after all the entries have been parsed we can set the global size and number of dimensions for trace_t. Once parsing has finished to conserve memory the input file buffer is freed from memory.

5.2 Naive Algorithms

Now that the structures have been populated our scheduler can begin rearrangement by passing the trace_t structure to the appropriate function selected from the scheduling algorithm command line argument with the following choices: random, sequential, round robin, and warp. All these options apart from warp are naive algorithms, meaning that they aren't efficient and don't aim to represent the scheduling from a real GPU since they don't take into account SIMT execution. Instead they are used to illustrate kernel access patterns through visualization.

5.2.1 Strategy

All these naive algorithms are initially setup in identical ways by first of all calculating the total number of threads, then creating an array of trace_entry_t structures for each thread. Each of which only stores the accesses for that thread and is allocated to the correct size by first of all finding the number of entries. Every thread also has an access counter initialised to zero that counts the number of memory accesses which it has already scheduled.

To aide cache simulation metadata is written to the first line of the output file before scheduling starts, see section 6.4 for details. This contains the warp width to help coalescing in the cache simulator, which for all naive algorithms is set to 1 since each access is intended to take place independently. The other metadata included is the number of workgroups calculated from equation 5.1.

$$\text{Number of Workgroups} = \sum_{d=1}^{Dimensions} \left\lceil \frac{\text{Global Size}_d}{\text{Local Size}_d} \right\rceil \tag{5.1}$$

All the individual thread structures are then scheduled together by iterating a loop for each element in the trace and selecting the thread to schedule the next access. On each iteration an access is written directly to the output file to eliminate complications from rearranging memory. The entry each thread schedules is pointed to by the thread's access counter which increments after each memory access is added.

Threads are not considered for scheduling however when the have either scheduled all their accesses, or when the trace entry currently pointed to by their counter is a

memory fence or barrier. Memory barriers are dealt with by only scheduling threads which have not yet reached the barrier, and once every thread has reach it then all the counters are incremented so that they point to the next access after the barrier. This preserves the barrier ordering, although it also enforces global constraints on less strict local synchronization.

5.2.2 Different Naive Polices

Three different naive scheduling policies are implemented: random, round robin, and random. For all these algorithms a variable is maintained which stores the index of the thread which made the last memory access, how this variable changes decides the scheduling policy.

In sequential scheduling every thread runs to completion, or until it sees a barrier, before the next thread is scheduled. Therefore the index variable only changes when these conditions are met by incrementing to schedule the next thread with decreasing priority the higher the dimension. In two dimensions this is equivalent to row wise accesses of a matrix.

For random scheduling however the next value of the index variable is changed on every iteration so that it points to another thread at random after each access. While in round robin every thread takes a turn making a single access so the counter is incremented on every iteration, and only reset to zero once every available thread has scheduled an access.

5.2.3 Example Output

In order to provide enough information for cache simulations each scheduled memory access written to the output file contains: the memory address, read/write, workgroup id, warp id, and instruction id. For a round robin scheduling example in listing 5.3 we know from the line of metadata at the top that warp width is one and there are four workgroups.

```
1 4
0x83BD000 R 0 0 1
0x83BD000 R 0 0 1
0x83BD000 R 0 0 1
0x83BD000 R 1 0 1
0x83BD000 R 1 0 1
0x83BD000 R 1 0 1
0x83BD000 R 1 0 1
0x83BD080 R 0 0 1
0x83BD080 R 0 0 1
0x83BD080 R 0 0 1
0x83BD080 R 1 0 1
0x83BD080 R 1 0 1
0x83BD080 R 1 0 1
```

Listing 5.3: Example Round Robin Scheduling Output. First line contains metadata: 'warp size' 'number of workgroups'. All entries have format: 'Memory address' 'Read-/Write' 'Workgroup' 'Warp' 'Instruction'.

5.3 Warp Scheduling

A more realistic scheduling algorithm aiming to simulate the Fermi GPU execution is warp scheduling. In warp scheduling instructions are executed in a SIMT pattern where multiple threads execute the same instruction concurrently. Warps are defined as a group of 32 threads that execute the same instruction simultaneously, although our scheduler can be configured to different warp sizes through the command line.

5.3.1 Warp Initialization

In order to recreate these warps when using warp scheduling our scheduler initially sets up warp structures which contain all the entries from threads within that warp. This allows accesses to be more easily scheduled both within individual warps and between warps.

The first step performed in warp scheduling is therefore initializing warp_t structures shown in listing 5.4 before any scheduling can begin. As warps can't span workgroups each warp is uniquely identified by two separate values, work_id, which identifies the workgroup containing the warp. As well as local_id to distinguish each warp from others in the workgroup. We can calculate the number of warps per workgroup as the ceiling of threads per workgroup divided by threads per warp, which will be constant across all workgroups and acts as an upper bound to local_id.

```
typedef struct warp_s
{
    trace_entry_t **entry;      //trace entries in warp
    t_dim_t work_id;            //workgroup id
    int local_id;               //id of warp within workgroup
    int warp_size;              //threads in warp, argument from user
    int length;                 //number of entries in warp
} warp_t;
```

Listing 5.4: Warp Structure

The second initialization step is to populate every warp with entries from it's constituent threads. To conserve memory the number of entries in each warp, length in warp_t, is calculated first so that the correct number of entries can be allocated. All the entries in the trace_t structure from listing 5.2 are subsequently looped over and copied to the appropriate warp structure. Before being freed from memory in trace_t since all scheduling will now take place using the warp structures. Each entry is assigned to a warp using it's thread id, as based on this id we can calculate the workgroup which the thread belongs by taking the thread id modulus workgroup size in each dimension. To then calculating the local id of the warp within that workgroup equation 5.2 is used which finds the local thread id and converts it to an offset from the first local thread. We can then divide this offset by warp size to get the local_id of the warp.

$$\text{Local id} = \frac{\sum_{d=1}^{Dimensions}((T_d \bmod W_d) \cdot \prod_{d'=1}^{d-1} W_{d'})}{\text{Warp Size}} \tag{5.2}$$

$$T_x = \text{Thread ID in dimension x}$$
$$W_x = \text{Workgroup Size in dimension x}$$

5.3.2 Intra-Warp Scheduling

Once our scheduler has assigned all the trace entries to a warp it's next task is to schedule all the entries within those warps so that instructions executed at the same time in the SIMT model are grouped together. In order to do this we create pointers for each thread pointing to the next thread access which hasn't been scheduled. Only accesses pointed to are ever considered for scheduling so that the execution order of each thread is preserved in the warp. When scheduling the accesses for the next SIMT instruction we first of all select candidate accesses as the entries which are pointed to by each thread, but only those that are not barrier calls. If all the threads point to barriers however then all the pointers advance, so preserving the cross barrier ordering by only advancing threads past barriers when everyone has seen them.

After we have all our candidate entries which could be scheduled next our warp algorithm identifies and schedules those candidates whose SIMT accesses were made first. In order to know which entries were executed first however we have to create a notion of time using our loop and instruction information to find the access with the instruction closest to the top of the most recent loop iteration. As instruction ids are

named statically in our trace transformation with increasing ids we can infer that the instruction with the smallest id will come before any others with the same loop iteration. Working out the most recent loop iteration however is more complex, recall from section 4.4 that every access has an associated loop iteration for up to three nested loops. Each of which is given a statically increasing label with the key property that a loop with a smaller label can't be nested inside another loop with a larger label value.

To get the candidate with most recent loop iteration our scheduler finds the accesses with the smallest loop label since this represents the outer most loop of any possible candidate loop nesting. Then only considering those candidates with the smallest iteration of that outer loop. No access not in this loop can come before this as otherwise it would need to have occurred before the loop, in which case it would have a smaller label. Or it would need to have happened at an earlier iteration, with a lower iteration value.

If several accesses have the same outer loop iteration then candidates are eliminated who are not in the lowest iteration of that outer loop. Before repeating the process by finding the next smallest loop label and it's minimal iteration. Possibly repeating once more for the last loop value if there is again more than one candidate left. In the case that more than one candidate has the exact minimal loop data then arbitrarily the first is returned.

Using the access calculated as coming from the next SIMT instruction executed any other candidates sharing the same instruction and loop iterations are scheduled together as they will be executed concurrently. The pointers for the threads which made these accesses are then advanced so the they point to the next thread entry to be scheduled. With this processes of scheduling the most recent accesses from the candidate entries repeating until all the accesses from the threads in the warp have been scheduled.

5.3.3 Inter-Warp Scheduling

From the previous section we have seen that thread accesses contained in a warp are scheduled together in the warp_t structure to recreate SIMT instruction flow by grouping all the access together for each concurrent execution. However once we have all our individual warps they need to be merged together so that the final arrangement can be written to a single output file, while retaining the correct execution order of SIMT accesses within the warps.

Each of the warps are therefore looped over in a round robin fashion taking turns to write accesses from a single SIMT execution to the output file. The way warps are indexed means that the workgroup the warp belongs to changes on every iteration, so there is also another level of round robin scheduling between the workgroups. On each iteration a single SIMT access is identified as all those accesses in the warp up to the limit of warp size which have the same loop iteration and instruction.

Chapter 6

Cache Simulation

Once we have the trace as rearranged by our scheduler we can finally estimate the kernel's cache performance. This is calculated by processing the trace using a cache simulation to reconstruct a single L1 Fermi cache. Our simulator is made as flexible as possible to represent any cache configuration by taking six command line arguments, as seen in figure 6.1. This chapter details how we adapted a generic cache simulator so that it could simulate parallel execution features like coalescing. In addition to how the subset of trace entries mapped to the SM containing our simulated L1 cache was chosen.

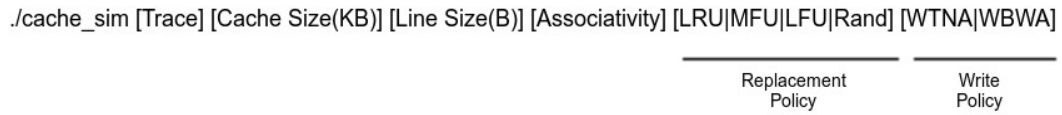


Figure 6.1: Command line arguments.

6.1 Modifications to Existing Simulator

Our L1 simulator is based on an existing cache simulator already available to us from a previous academic assignment [29]. This original simulator was for a generic single layer cache that could be set to different sizes and associativities before returning the miss rate as the lone statistic. Therefore the additions we had to make are:

- `More Thorough Statistics` Section 6.2
- `Input Format` Section 6.3
- `Filtering of Certain Workgroups to Process` Section 6.4
- `Imitation of Coalescing` Section 6.5

6.2 Statistics

```
// Structure for holding stats about cache performance.
typedef struct stats_s
{
    int reads;                          // Number of reads
    int readMisses;                     // Number of read misses
    int writes;                         // Number of writes
    int writeMisses;                    // Number of write misses
    int writeBacks;                     // Number of write backs
    int coldMisses;                     // Number of cold misses
    int capacityMisses;                 // Number of capactiy misses
    int conflictMisses;                 // Number of conflict misses

    std::stack<stack_t> *stack;         //cache line reuse distance stack

} stats_t;
```

Listing 6.1: Cache statistics structure

To get this more comprehensive performance data a statistics structure is maintained, see 6.1, which we update after each cache access using a reference stored in the cache structure. The number of write backs refers to a situation in the write back, write allocate policy where a dirty cache line is evicted from it's set and the modified value needs to be written back to memory. Each read miss is also categorized as either cold, capacity, or conflict using a reuse distance stack from Beyls & D'Hollander [8]. Where for every cache line accessed the stack is checked to measure the line's distance from the top, before popping that line and pushing it back to the top. If the reuse distance is greater than the number of lines in the cache it is a capacity miss, otherwise it is a conflict miss, and if the line doesn't exist at all in the stack then it is a cold miss.

6.3 Input

Before modification the existing simulator took input by being called directly like a library for each read and write by the program being investigated. This was changed to reading an input file which could be output from our scheduler containing all the accesses. Each element in our input contains a workgroup id, warp id, and instruction id in addition to the memory address and read/write data that a normal cache simulator would use so that coalescing can be reconstructed, listing 5.3.

In order for our cache simulation know the size of a warp it is parametrized so that the value can be read from the first line of the input trace provided by the scheduler. This first line also contains the total number of workgroups so the the simulator can calculate how many workgroups to allocate the SM our simulated L1 cache is located on.

6.4 Filtering Workgroups

As L1 caches are local to a single SM core only a subset of workgroups will be assigned to each SM, and so be seen by our L1 cache. Therefore our L1 simulator first of all needs to choose workgroups which it will process, and then execute the accesses from only those workgroups. Before we can assign workgroups to our L1 cache however the simulator works out how many workgroups we need to allocate. Workgroups per SM simply equals the number of workgroups divided by number of cores, then rounded either up or down if they are not divisible. As a result we chose to simulate the ceiling of this value but our profiler may profile a core which rounds this down, requiring repeated profilings to normalize results. Unfortunately this value depends on the number of SM cores on the GPU hardware simulated, which is a machine dependent factor. Since the machine we are using for validation has a NVIDIA GTX480 [18] GPU with 15 active cores this is the value our simulator is configured to use.

Once we know how many workgroups our simulator has to allocate we need to pick workgroups to assign. Tang et al [28] make the assumption that there is low probability of adjacent blocks being mapped to the same core, resulting in little cache data reuse between workgroups assigned to the same core. We use this idea to assign workgroups to our cache randomly, as long as there are no duplicates. However because workgroups are assigned dynamically in hardware when comparing simulated results against profiler results we need to perform repeated profilings. As for different runs there may be differing degrees of inter workgroup cache reuse depending on how the workgroups were assigned to cores both in hardware and our simulation. Once we have defined the workgroups we are going to process our simulation executes the trace for every workgroup, using the workgroup id in the input to only simulate accesses from the selected workgroups.

6.5 Coalescing

Due to the GPU parallel execution model changes also had to be made to allow the simulation to reconstruct parallel cache accesses from a sequential input trace. Coalescing in GPU hardware allows an entire cache line to be read/written in a single transaction for all the threads in a warp accessing the same cache line. This concept motivates Fermi's choice of 128 byte cache line size in both the L1 and L2 levels since warps contain 32 threads. So when accessing 4 byte data elements like integers or floats all the threads can map to the same cache line. If threads in the same warp access different cache lines however then the accesses need to be serialized, leading to stalls. This can result in one or more threads in the warp having a miss in the cache, which is handled by the GPU making the warp inactive and scheduling another warp to hide the latency.

To then imitate coalescing for each workgroup we use the warp size parameter from the input metadata as the upper limit to a counter. This counter increments on every access and resets to zero when it reaches the warp size, or an access not from the current warp is made. An access not from the current warp can either be an access by a thread in the

workgroup with a different warp id. Or an access by a thread in the warp to a different instruction than the rest of the warp is accessing. Warp id and instruction id are added by the scheduler to each entry of the input for identifying these scenarios.

By comparing the value of this counter against stack reuse distance we can then determine if the addressed line has been seen before in this coalesced access. The comparison between stack distance and reuse distance works, since if the counter is at X then it means that this is the Xth thread in the warp making an access. Therefore if reuse distance is less than X we know that the line has been accessed before by this warp. If the lines has indeed been seen before then the line is not processed or statistics updated, since this access would have been coalesced into the earlier access. Conversely if the line has not been seen before in this warp then the access is treated as a new coalesced access which would be serialized in hardware.

Chapter 7

Experimental Setup

In this chapter we outline the configuration of our simulation and the hardware platform it's validated against to show that the results achieved are realistic. The benchmarks used for comparison are also described to illustrate their properties and highlight why they are useful for cache analysis.

7.1 Validation

In order to validate the accuracy of our model we need to compare our simulated results against those obtained from hardware counters on a real Fermi GPU. These hardware counters are taken from running the same kernel as simulated but without instrumentation on a NVIDIA GTX480 [18] GPU with OpenCL v1.1. Using NVIDIA's compute command line profiler [15] from CUDA Toolkit [16] version 4.2.1 to monitor the activity of a single L1 cache. Other NVIDIA profilers were available but the command line profiler was chosen since it supports OpenCL, whereas others like NVPROF [20] only support NVIDIA's own CUDA [21].

The specific counters that the profiler is configured to record are l1_global_load_hit and l1_global_load_miss. Defined as the number of lines which hit/miss in the L1 cache for global memory load accesses, which in the case of perfect coalescing increments by 1 for warp accesses to 4 byte data objects. All profiler counters were taken using the 16 KB L1 cache configuration on the GTX480, as opposed to 48 KB, giving the cache 32 4-way sets of 128 byte lines.

7.2 Simulation Configuration

The whole trace based simulation including instrumentation, OpenCL kernel execution, scheduling, and cache simulation used to gather results was run on an Intel Pentium T2390 [6] dual core 1.86GHz CPU. Since the OpenCL kernels are running on a CPU there is minimal parallelism and so the simulator is more thoroughly tested since

it has to rearrange an almost sequential trace into a parallel one. OpenCL v1.2 is run using the AMD APP SDK v2.8 [3] platform which also supports Intel processors as both AMD and Intel use the x86 ISA.

LLVM version 3.3 is used for the instrumentation process, which involves using clang with optimization level -O0 to compile the kernel into IR without any optimizations at all. The LLVM optimizer tool is then run to perform the instrumentation using the -S option to output IR that can be converted back into OpenCL with Axtor. The optimizer also runs the mem2reg [11] library transformation which promotes memory references to register references in order to facilitate our custom transformations.

Our cache simulator can be configured with different cache configurations through the command line, see figure 6.1. All our experiments are therefore run to replicate the Fermi 16KB cache configuration which has 128 byte lines in 4-way associative sets, with a LRU replacement policy and write through, no allocate write policy.

7.3 Profiler Measurements

Profiler results are taken from the median of 100 kernel runs on hardware since counters change from run to run depending on the number of workgroups assigned to the profiled core, as well as the amount of cache reuse between them. Median is used instead of mean to mitigate the effect of outliers on the data.

Our simulation is also run multiple times for each kernel as our cache simulation picks workgroups at random to simulate, with differing degrees of data reuse between them. The median of 20 runs through the cache simulation is used as the final value.

The relative standard deviation of each benchmark is presented to illustrate the variation between runs, both for our simulation and the profiler counters. This is equivalent to the standard deviation over the median, multiplied by 100 to give a percentage. A higher percentage implies higher variance while a percentage of zero means all the samples are identical.

We define the miss rate as a percentage of misses from total number of reads, representing the rate of expensive off SM cache misses. The percentage error defined in equation 7.1 is used to determine the accuracy of our simulated results.

$$\text{Percentage Error} = 100 \cdot \left| \frac{X_{profiled} - X_{simulated}}{X_{profiled}} \right| \tag{7.1}$$

7.4 Benchmarks

Three popular OpenCL kernels are used for experimentation: matrix multiplication, matrix transposition, and stencil. These three benchmarks were chosen as their access

patterns are easy to understand and therefore cache performance can be predicted theoretically. Additionally all these benchmarks have large amounts of spatial and temporal locality, meaning the cache will be accessed frequently and the impact of any changes to parameters will be more obvious.

However our experiments are constrained to some degree by the maximum workgroup size of 1024 in the GTX480, as well time and memory limitations running the simulator. Any increased simulator resource consumption mainly results from a large number of threads making minimal accesses, which is computationally more complex to process than a few threads making a large amount of accesses.

7.4.1 Matrix Multiplication

Matrix multiplication, listing 7.1, uses a two dimensional thread space to calculate the resulting matrix C from the multiplication of two input matrices A & B. Every thread calculates and stores a single element in the output matrix from the results of reads to the other two input matrices, all of which are to global memory.

In parallel matrix multiplication a thread sums the products of all the elements in a row of one input matrix against all the elements in a column of the other matrix. In this kernel matrix A is being accessed by row and matrix B by column. Since matrices are stored by row in memory, we would expect matrix A to be accessed stride 1 by each thread, and matrix B by a stride equal to row width. There is a lot of data reuse in matrix multiplication since rows and columns are typically referenced several times, exhibiting temporal locality, and the elements in each row are also stored close together, so also preserving spatial locality. These are the two properties that caches are designed to take advantage of and so the large amount of cache traffic provides more interesting cache results than simply cold misses.

```
__kernel void mm(const __global float* A,
                 const __global float* B,
                       __global float* C,
                 uint width, uint height) {
  uint row = get_global_id(1);
  uint column = get_global_id(0);

  float tmp = 0.0f;
  for (uint index = 0; index < width; ++index){
    tmp += A[row * width + index] * B[index * width + column];

  }
  C[row * width + column] = tmp;
}
```

Listing 7.1: Matrix multiplication kernel

7.4.2 Matrix Transposition

Our matrix transposition program in listing 7.2 takes an input matrix idata and performs transposition on it to populate a separate matrix odata. Each thread processes a single element from the input matrix and copies it to the transposed location in the output matrix. Again this is done using a two dimensional thread space with the same number of threads as matrix elements, where every thread makes a only one read and one write to global memory.

Since the cache is write through, no allocate the writes to the output matrix shouldn't affect the cache as they go straight back to main memory, and aren't loaded back in with a follow up read. Additionally since the writes are to a separate data structure that is never read, no data from the odata matrix will ever be in the cache. Therefore it is only the idata matrix which influences the cache, with every thread making a single read to a unique matrix element. Due to the simplicity of the kernel, we expect to predict the miss rate exactly for all matrix sizes with our simulator.

```
__kernel void mt(__global float *odata, __global const float* idata,
    int width, int height) {
  unsigned int row = get_global_id(1);
  unsigned int column = get_global_id(0);

  unsigned int indexIn  = row * width + column;
  unsigned int indexOut = column * height + row;
  odata[indexOut] = idata[indexIn];
}
```

Listing 7.2: Matrix transposition kernel

7.4.3 Stencil

Stencil is a parallel programming pattern carried out iteratively, where values are updated based on the elements at relative offsets to the value's location. The group of inputs which contribute to the value being written is called the neighbourhood.

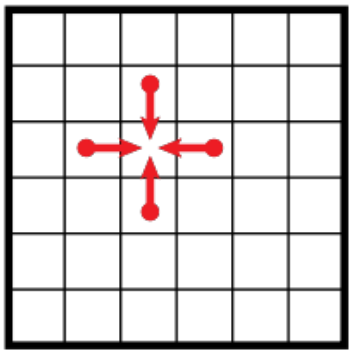


Figure 7.1: Jacobi method cell neighbourhood in two dimensions. [Source:"http://en.wikipedia.org/wiki/File:2D_von_Neumann_Stencil.svg"]

The stencil pattern is useful for a number of applications, such as particle simulations and image processing, however the algorithm our kernel applies in listing 7.3 is the Jacobi method. In the Jacobi method elements are typically updated iteratively in passes until all the values have converged using a neighbourhood of adjacent values. In two dimensions the neighbourhood of a cell is illustrated in figure 7.1, although edge values are typical not updated in Jacobi since they have less neighbours and instead act as a border to help update other cells.

Our stencil kernel however operates in three dimensions, directly translating to a three dimensional thread space where each thread updates a single element in the stencil based on the adjacent cell in all three dimensions. Here each cell is updated to it's new value in output array Anext based on constant c1 times the summation of the previous adjacent values stored in A0, minus it's own previous value times constant c0.

The output matrix is therefore written to once on every iteration by each thread as in earlier examples. All the adjacent values in the stencil will also be read once, totalling 7 reads per thread since the cells previous value is also read from A0. The spatial locality of this kernel means that it should provide interesting cache results with lots of data reuse, added to by the temporal locality from subsequent iterations.

```
#define Index3D(_nx,_ny,_i,_j,_k) ((_i)+_nx*((_j)+_ny*(_k)))

__kernel void naive_kernel(float c0,float c1,__global float* A0,
    __global float *Anext,int nx,int ny,int nz)
{
       int i = get_global_id(0)+1;
       int j = get_global_id(1)+1;
       int k = get_global_id(2)+1;

       if(i<nx-1)
       {
           Anext[Index3D (nx, ny, i, j, k)] = c1 *
           ( A0[Index3D (nx, ny, i, j, k + 1)] +
             A0[Index3D (nx, ny, i, j, k - 1)] +
             A0[Index3D (nx, ny, i, j + 1, k)] +
             A0[Index3D (nx, ny, i, j - 1, k)] +
             A0[Index3D (nx, ny, i + 1, j, k)] +
             A0[Index3D (nx, ny, i - 1, j, k)] )
           - A0[Index3D (nx, ny, i, j, k)] * c0;
       }
}
```

Listing 7.3: Stencil kernel

Chapter 8

Results

In this chapter we compare our simulated results against those from hardware using the experimental setup described in the previously chapter.

For experiments using our matrix computation benchmarks we varied both the global and local size of the matrices while keeping the matrix square. Table 8.1 shows the relationship between these two factors where the left column has the workgroup size x the number of workgroups for a single dimension. While the right column shows the overall size of the matrix calculated using equation 8.1.

Workgroup Size x Number of Workgroups	Matrix Size
16 x 2	1024
16 x 3	2304
16 x 4	4096
32 x 2	4096
16 x 5	6400
16 x 6	9216
32 x 3	9216
16 x 7	12544
16 x 8	16384
32 x 4	16384
16 x 9	20736
16 x 10	25600
32 x 5	25600
16 x 16	65536
32 x 10	102400

Table 8.1: Table of workgroup configurations against matrix size.

$$\text{2D Square Matrix Size} = (\text{Workgroup size in 1 Dim} \cdot \text{Number of workgroups in 1 Dim})^2 \tag{8.1}$$

8.1 Matrix Transposition

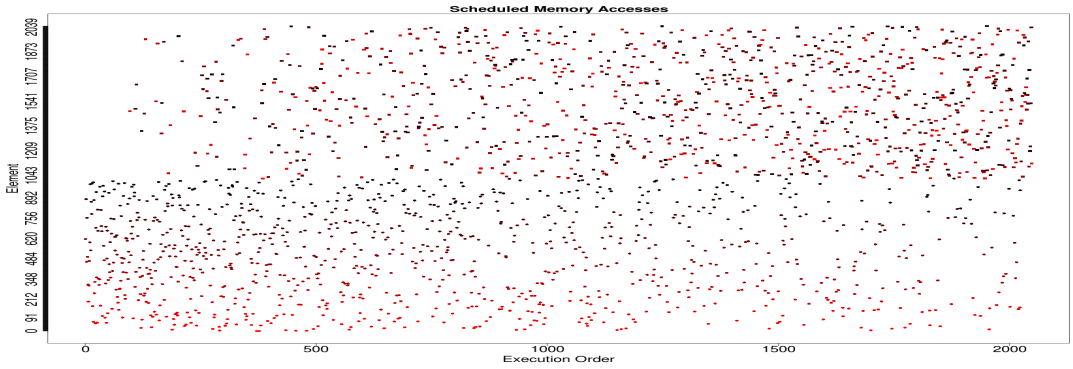
Matrix transposition is a benchmark with no data element reuse and as a result we can predict it's performance exactly with our simulator since every cache access should be a miss.

8.1.1 Visualization

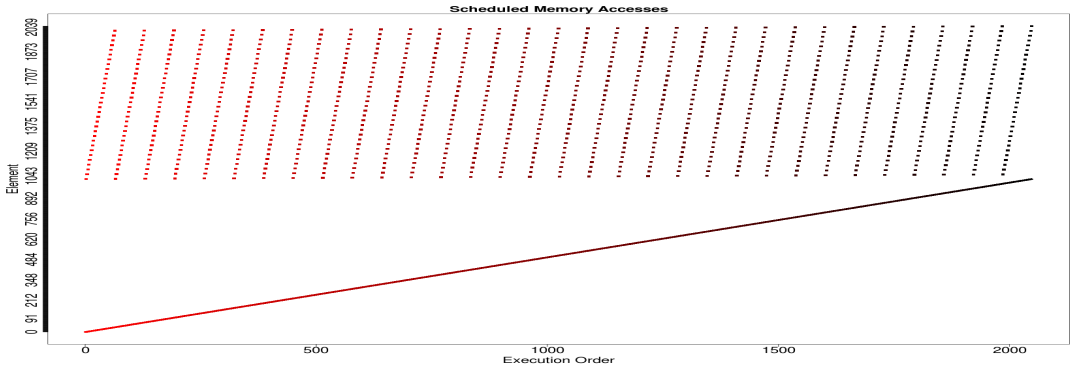
In order to visualize our scheduled trace the scheduler outputs another version of the trace in exactly the same order as input to the cache simulator but formatted so that it can be used by R [30] graphing software. We can then visualize the memory access pattern using R to produce a plot of all the memory locations accessed over time. Memory locations are plotted on the y-axis and represent the element offset calculated by subtracting the memory address from the lowest memory address, and then divided by 4 since this is assumed to be the size in bytes of a data object. Execution order from the file is then plotted on the x-axis to represent time, note that these accesses aren't coalesced since this is done later by the cache simulator. The colour of each point identifies the thread which made the access, with threads that are close together in the first dimension having similar colours.

By visualizing the trace of matrix transposition, figure 8.1, using 32 x 32 matrices with several scheduling policies we can clearly identify the two matrices from their access patterns as well as the data structure boundary of 1024 elements. We can infer the input matrix as being the bottom matrix, elements 0-1023, and the output matrix as being the top matrix, elements 1024-2047. The input matrix being read is accessed in row major order with thread id in the first dimensions, leading to a stride of one between elements. While the transposed matrix being written uses the first dimension thread id to reference the column, so there is a stride 32 reference pattern since this is the matrix width.

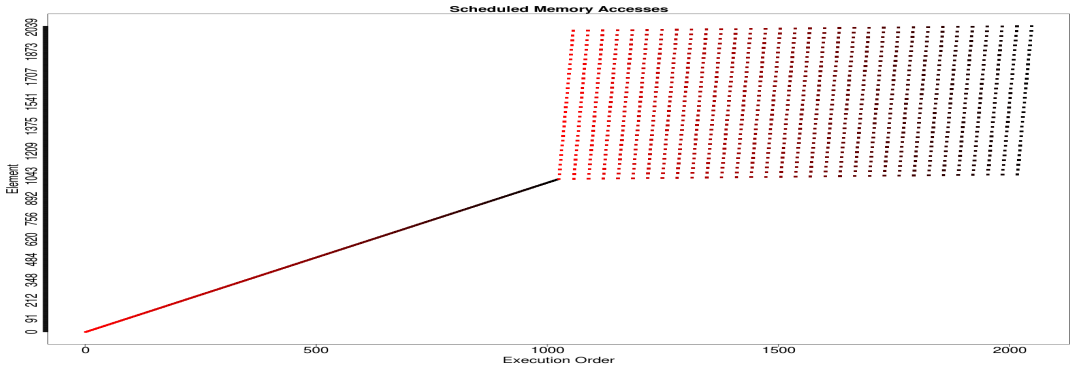
We can also see from the visualizations the effect of the different scheduling policies. With sequential for example, figure 8.1b, the writes from each of the 32 threads are clearly visible as 32 vertical lines with a distinct colour which gradually changes as each thread is scheduled. Round robin scheduling in figure 8.1c however exhibits a clear vertical split between the two instructions resulting from all the reads being scheduled before any of the writes. Warp scheduling in figure 8.5d looks slightly different though as although it shows the vertical split from round robin scheduling there is further grouping of accesses from the workgroup configuration of 16 x 2. This results in the formation of 8 warps per workgroup, so scheduling 32 accesses from a warp before switching to the next of the four workgroups to schedule a further 32 accesses from another warp. Alternating workgroups causes clusters of four 32 element accesses that can be seen most clearly in the bottom input matrix. Where two workgroups warps with similar colours make accesses to a higher up location, with the warps from the other two workgroups making access to lower addresses. Each warp is illustrated best in the top output matrix as a two column block of 32 accesses with the same colour.



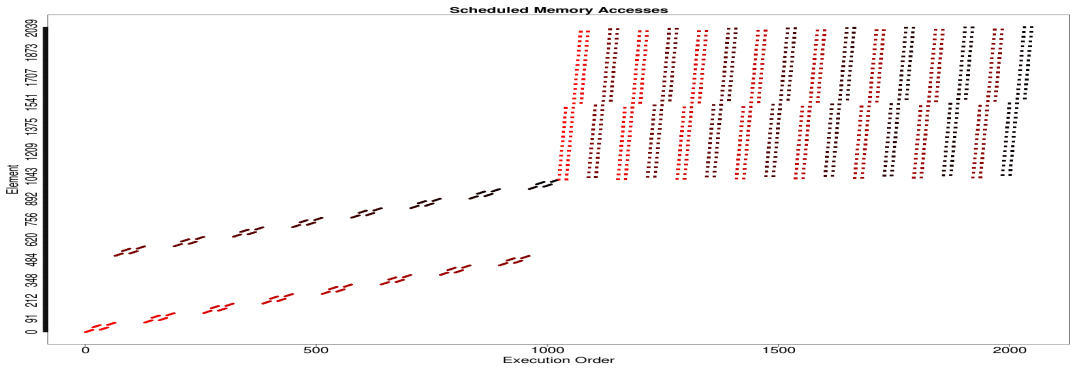
(a) Random scheduling



(b) Sequential scheduling



(c) Round Robin scheduling



(d) Warp scheduling

Figure 8.1: Visualization of different scheduling algorithms on the matrix transposition benchmark with a 16 x 2 workgroup configuration

8.1.2 Experiments

In order to assess the success of our simulation we validate our results using the warp scheduling algorithm against the hardware described in the experimental setup, chapter 7. This involves taking the median of 100 runs of the profiler on the benchmark with the same configuration as simulated, since the results can change on different runs due to the dynamic mapping of workgroups to SMs.

The relative standard deviation of these results is show in figure 8.2 for different workgroup configurations. Here we can see that once the number of workgroups reaches 100 the variance increases. This is because more than one workgroup is mapped to each of the 15 SMs resulting in different amounts of data reuse depending on the assignment. Additionally variance occurs from differing numbers of workgroups simulated between runs, as the number of workgroups is not perfectly divisible by the 15 SMs. Separate profiling also have to be done for reads and misses since both can't be profiled at once which is why each has separate standard deviations.

Our simulations results also exhibit variance as the workgroups to simulate are chosen at random, leading to different amounts of inter-workgroup cache line reuse. This is only prevalent in misses though, as the same number of cache reads will be made regardless as accesses are perfectly coalesced.

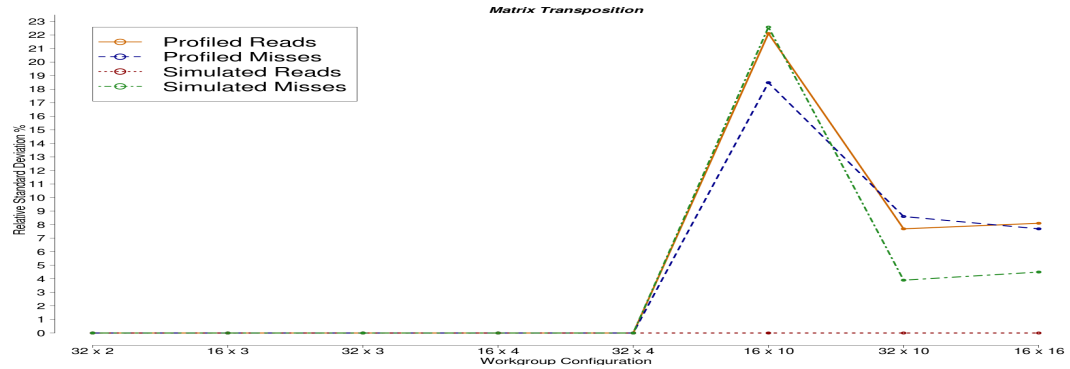


Figure 8.2: Relative standard deviation of results for matrix transposition kernel with different workgroup configurations

Experiments were done with a range of workgroup configurations up to a total matrix size of 102400, with results for number of reads and read misses shown in figures 8.3 & 8.4 respectively. Our simulator almost perfectly predicts both the number of reads and read misses, the worst results are for the 16 x 10 configuration which is 16 above the 96 profiled reads and misses. Still within the profiling absolute standard deviations of 21.2 for reads and 18.5 for misses. The most important aspect of our results however the that we have simulated the profiled miss rate of 100% for all configurations since every cache read is a miss.

We can derive that every read will be a cache miss since perfect coalescing allows a whole cache line to be read at once because the matrix being read is accessed with a stride 1 reference pattern. With every cache line corresponding to at least one matrix row as a block has enough space for 32 integers. This cache line is then never used

again when an uncached matrix row goes on to be transposed as no element in the same matrix is ever accessed more than once. Meaning that there can never be another access which hits that line.

For the column wise writes to the transposed matrix coalescing would mean that many different cache lines may be accessed, but because these are writes they don't impact the L1 cache. This is since the Fermi L1 cache is write through, no allocate meaning that the new data is written straight back to memory and the new value is not loaded into the cache.

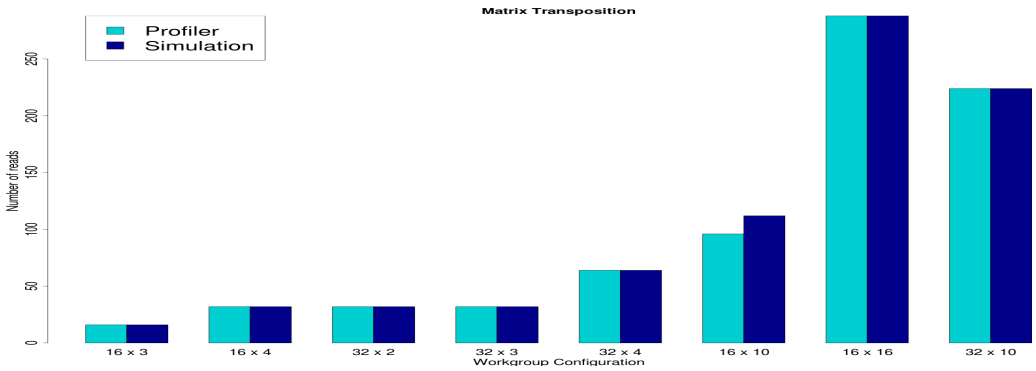


Figure 8.3: Number of reads for matrix transposition kernel with different workgroup configurations

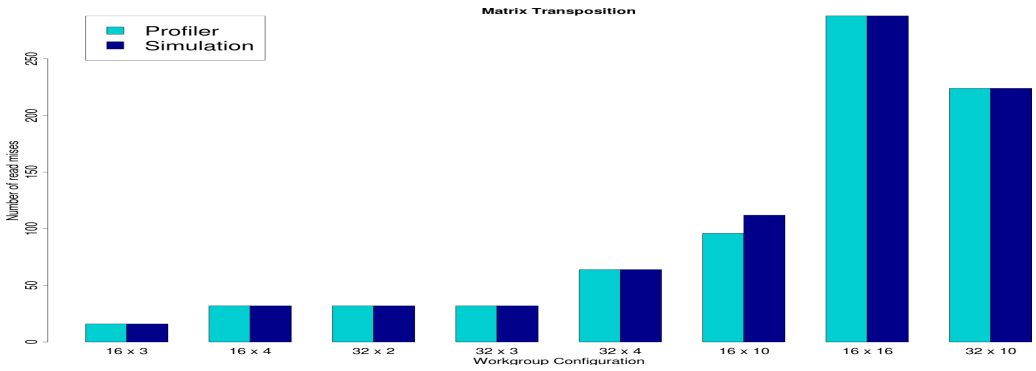


Figure 8.4: Number of read misses for matrix transposition kernel with different workgroup configurations

8.2 Matrix Multiplication

The matrix multiplication benchmark has large amounts of spatial and temporal locality causing it's utilization of the cache to be crucial to performance as it features more data reuse than matrix transposition. Our results show that we can accurately predict the number of reads. Read misses however are underestimated in our simulation when the number of workgroups exceeds the hardware limitations of workgroups per SM, as this causes the extra workgroups to wait until another workgroup has finished. Even

with this discrepancy though the miss rate is still calculated reasonably accurately to within 6%.

8.2.1 Visualization

Using R to plot the scheduler output we can visualize the effect different scheduling algorithms have for matrix multiplication with 6 x 6 matrices in a 3 x 2 configuration, to give us 36 elements in each matrix and therefore 36 threads.

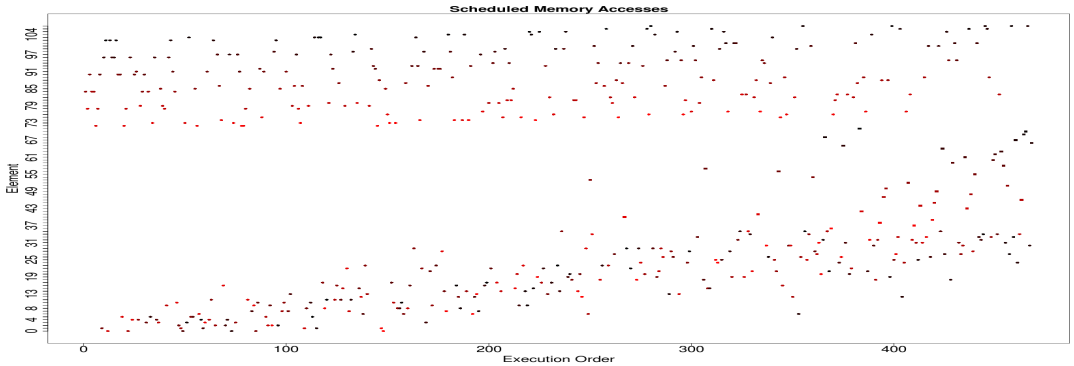
The first matrix in figure 8.5, elements 0-35, corresponds to input matrix B in the kernel, listing 7.1, since it is referenced in row major order using thread id. Meaning that adjacent threads in the first dimension of the thread space reference adjacent matrix elements in a single row on each loop iteration. The product output matrix C can be identified as having elements 36-71 due to only making one access per thread. The remaining matrix A consists of the final elements 72-107 which maps thread ids to columns in the matrix, so that all the threads with the same id in the first dimension in the thread space access a single element per iteration.

The random scheduling algorithm, figure 8.5a, produces more interesting results than the same algorithm in matrix transposition, as there is a pattern arising from the fact that every read must be made by a thread before it can write it's final value. This causes the absence of accesses for matrix C until around half way where some threads begin to finish.

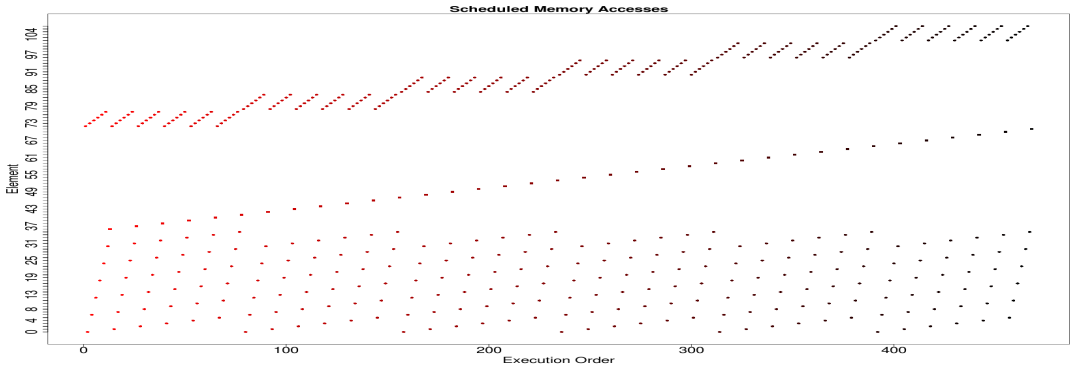
From figure 8.5b we can also see 36 vertical lines with a gradual colour change from sequential scheduling as each line corresponds to a thread. Spacing between accesses is much closer in matrix A since the loop iteration increments the location rather than being a multiplicative factor like matrix B.

The effect of thread id of matrix addressing is more apparent in round robin scheduling however, figure 8.5c. Where in matrix A we can see how threads with the same id in dimension one access the same element, with six groups for each of the six different first dimension ids. In matrix B however however all the threads which have the same id in the second dimension hit the same memory location, and due to round robin cycling through threads by first dimension this is every sixth access.

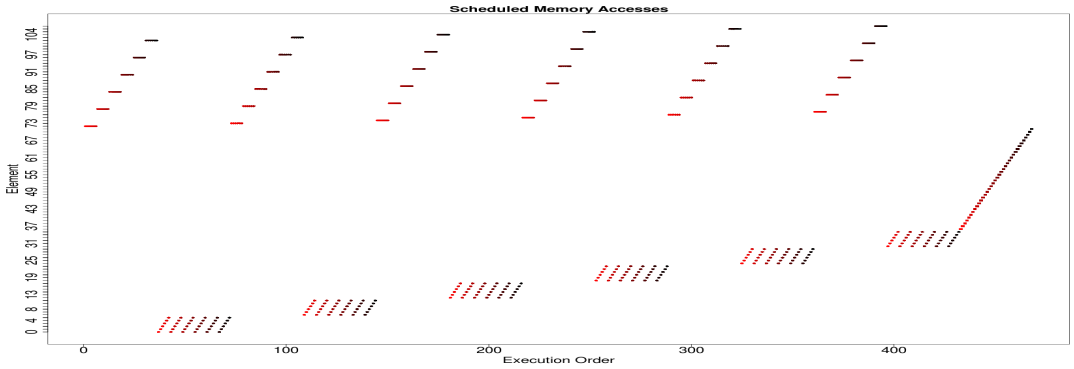
Warp scheduling in figure 8.5d again looks different due to the workgroup configuration of 3 x 2 being taken into consideration, resulting in one warp from each of the four workgroups of 9 threads. These warps can clearly be seen in matrix B as a square cluster of accesses. In matrix A these 9 accesses cluster into three groups of three hitting the same element. While in matrix C each group of three hits consecutive elements before jumping by three to the next group.



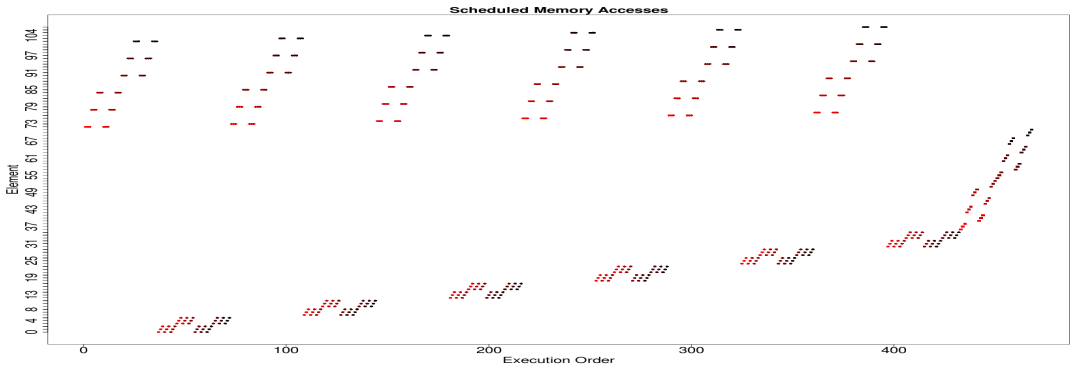
(a) Random scheduling



(b) Sequential scheduling



(c) Round Robin scheduling



(d) Warp scheduling

Figure 8.5: Visualization of different scheduling algorithms on the matrix multiplication benchmark with a 3 x 2 workgroup configuration

8.2.2 Experiments

We compared read and read miss results from our simulation against those from the command line profiler for several workgroup configurations up to a matrix size of 25600. The relative standard deviation of profile counters shown in figure 8.6 is erratic, overall with larger workgroups exhibiting more variance than smaller workgroups. Variance only appear appears when number of workgroups is greater than the 15 SM cores and for the smaller local size of 16 tends to increase with number of workgroups. The relative standard deviation in our simulated misses however results from workgroups being chosen at random. Only once we have to choose more than one workgroup does this become a factor due to varying amounts of data reuse depending on workgroups chosen.

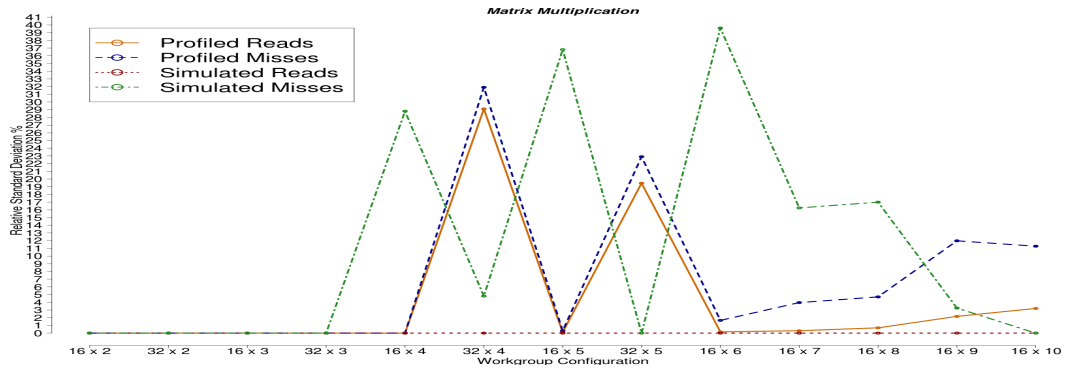


Figure 8.6: Relative standard deviation of results for matrix multiplication kernel with different workgroup configurations

From figure 8.7 which presents the number of reads we can see that our simulator can accurately predict the number of reads for all workgroup configurations, with a percentage error never exceeding 0.1 even for configurations with high standard deviation. This is due to perfect coalescing.

Our simulators estimation of misses in figure 8.8 however deviates by a percentage error of up to 47% for configurations with large numbers of workgroups. Configurations with less than 64 workgroups are better predicted by our simulator though. As the number of misses is within a percentage error of 19% for the 16 x 6 worst case, although our simulation has high relative standard deviation for this configuration of 39%. After this point however the number of misses is underestimated.

This discrepancy is because the GTX480 [18] hardware profiled has a compute capability of 2.0 [19], meaning that it is classified to meet certain computational specifications [22]. One of these is that the number of workgroups mapped to a SM at once cannot exceed 4, and as there are 15 active SMs in the GTX480 this means that the number of active workgroups cannot exceed 60. Any extra workgroups are stalled and then assigned dynamically when another workgroup terminates to free a SM slot, a feature not currently supported by our simulation. This causes the jump in miss rate from the 16 x 8 miss rate onwards as there becomes more than 60 workgroups and additional conflict misses are introduced. As when the extra workgroups are scheduled

they evict data that is currently being used, replaced with cache lines that all the other workgroups have previously finished with.

The miss rate in figure 8.9 reflects this increase in number of misses when there are more than 60 workgroups, as otherwise miss rate remains relatively constant at 6% but doubles to almost 12% after the threshold which we don't predict. Although at worst we only simulate 5.3% under the profiled miss rate of 11.7%.

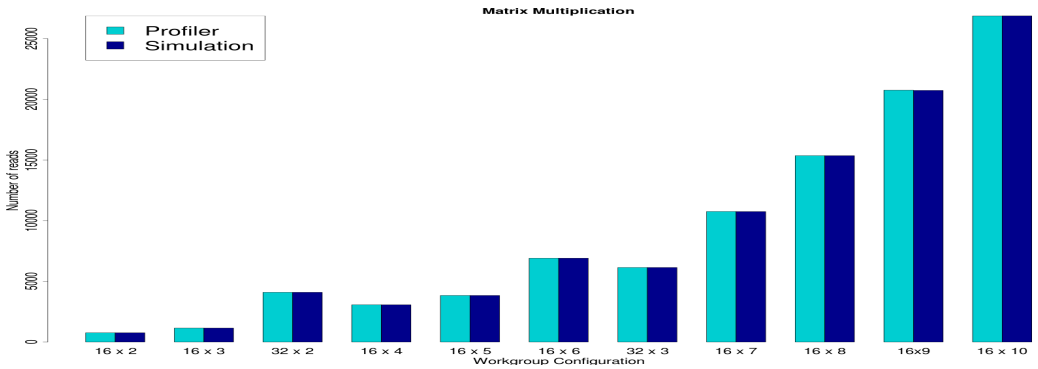


Figure 8.7: Number of reads for matrix multiplication kernel with different workgroup configurations

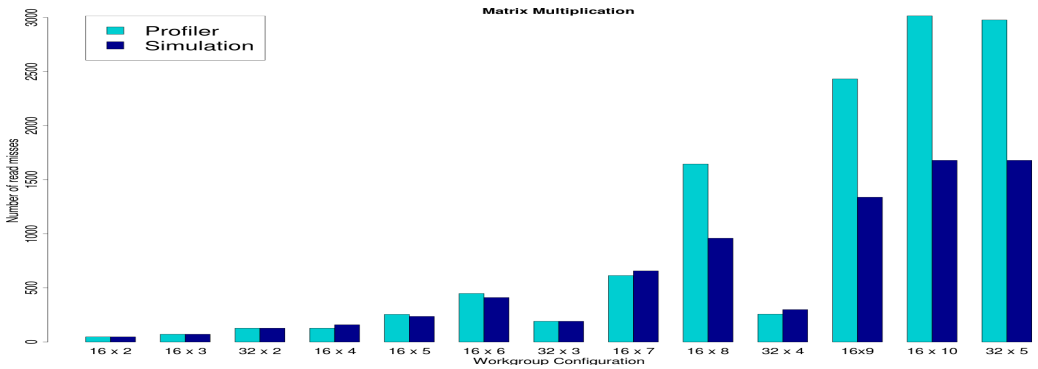


Figure 8.8: Number of read misses for matrix multiplication kernel with different workgroup configurations

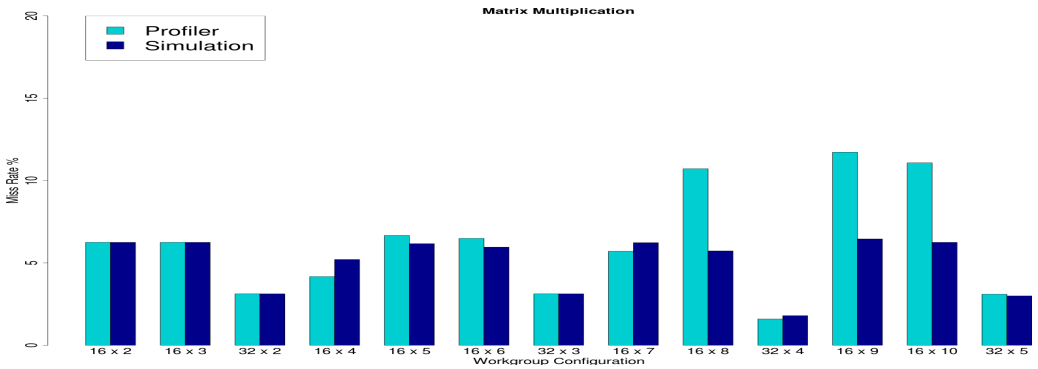


Figure 8.9: Miss rate for matrix multiplication kernel with different workgroup configurations

8.3 Stencil

Experiments were also carried out on the stencil benchmark, listing 7.3, in only one configuration. This is because the program takes an input binary file to populate data in the array, and since the benchmark was modified from the Parboil [27] benchmark suite the only file provided was for a 128 x 128 x 32 dimensional matrix. Since the edge values don't update in Jacobi this results in a thread space of 126 x 126 x 30, totalling 476,280 threads. Although Jacobi is an iterative algorithm due to its large size only one iteration was run because of time and memory limitations in the scheduler. However there is still a large amount of data reuse to provide meaningful cache results because of spatial locality. Our simulation results show that despite the issues with large numbers of workgroups discussed in the previous section we can still predict the miss rate within 2% of the profiled results.

8.3.1 Visualization

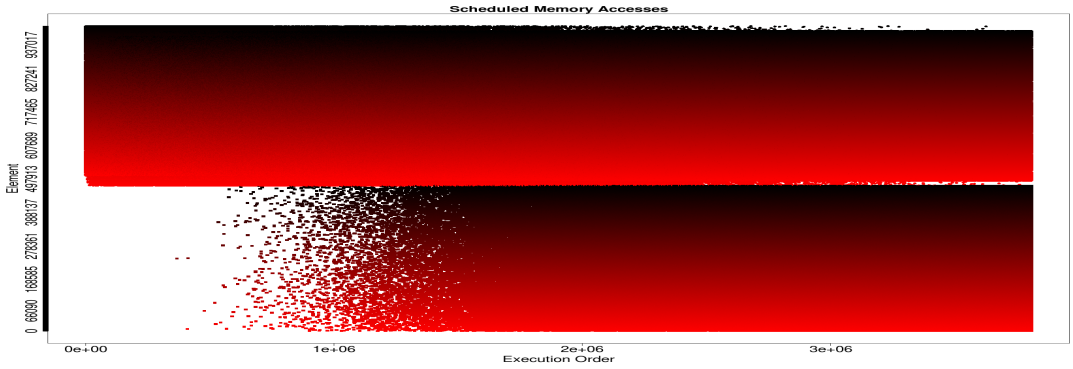
When visualizing the trace of the stencil benchmark in figure 8.10 we can see that the bottom structure, elements 0-476279, corresponds to the output matrix 'Anext' since there is only one access by each thread corresponding to the write. The top elements 476280 - 952559 therefore correspond to the A0 matrix being read from.

In the random scheduling algorithm, figure 8.10a, the delay from each thread having the make all 8 reads from memory before issuing the write postpones the start of the writes to 'Anext'.

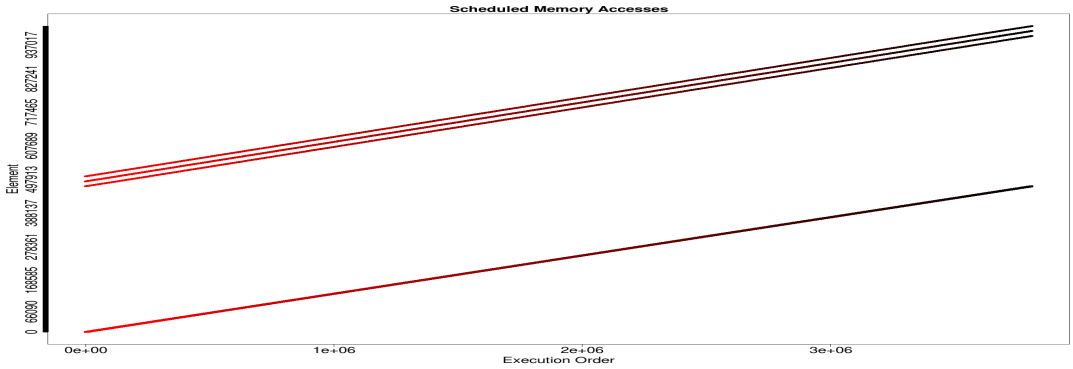
However figure 8.10b shows some interesting results for sequential scheduling as each thread makes reads to 3 groups of addresses, corresponding to the three different dimensions being referenced.

Round robin scheduling in figure 8.10c has a vertical line of accesses for each memory instruction, 7 reads and one write. An interesting feature of the reads however is that the first access is shifted up higher than the last 5, while the second access is shifted down lower than the last 5. This is because these two accesses are to adjacent elements in the third dimension which will be stored furthest away, causing a visible jump in address.

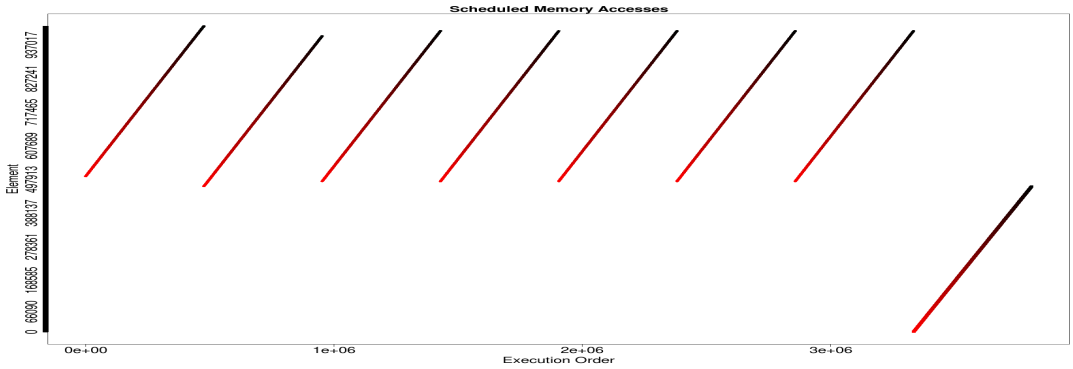
This effect is seen in warp scheduling as well, figure 8.10d, where there are two columns for each access as each workgroup contains two warps. Therefore round robin scheduling between workgroups causes every workgroup to schedule one warp before the other warp from the same instruction can be scheduled.



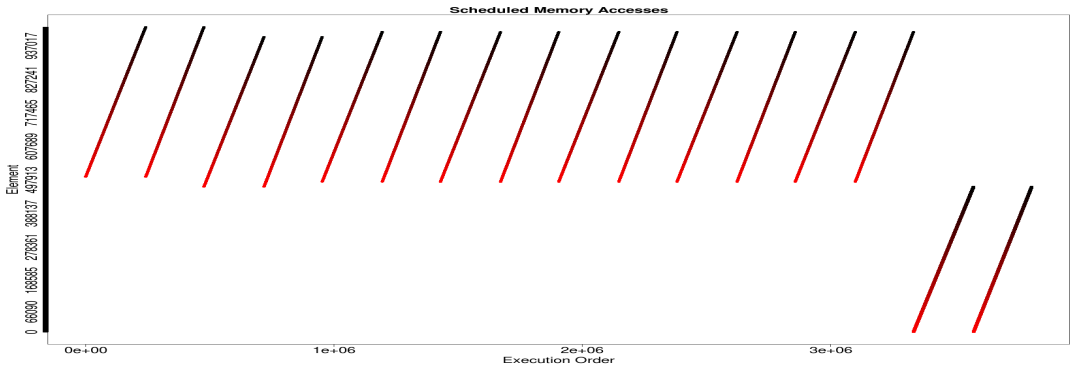
(a) Random scheduling



(b) Sequential scheduling



(c) Round Robin scheduling



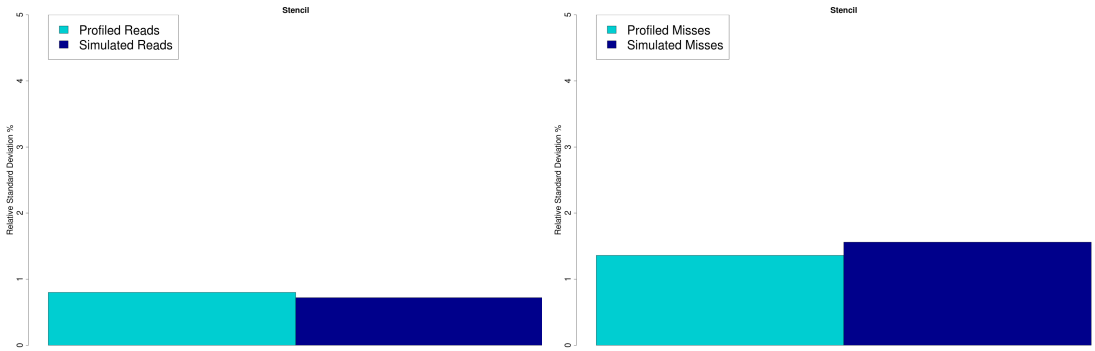
(d) Warp scheduling

Figure 8.10: Visualization of different scheduling algorithms for the 128 x 128 x 32 stencil benchmark

8.3.2 Experiments

Despite our sole configuration containing 7560 workgroups from a 64 x 1 x 1 workgroup size the relative standard deviation of profilings from the stencil benchmark is relatively low, figure 8.11, at under 2% for both reads and misses. This is because over the 15 cores exactly 504 workgroups are assigned to each SM, although only four can be active at once. Therefore there is no variation in number of workgroups profiled between runs.

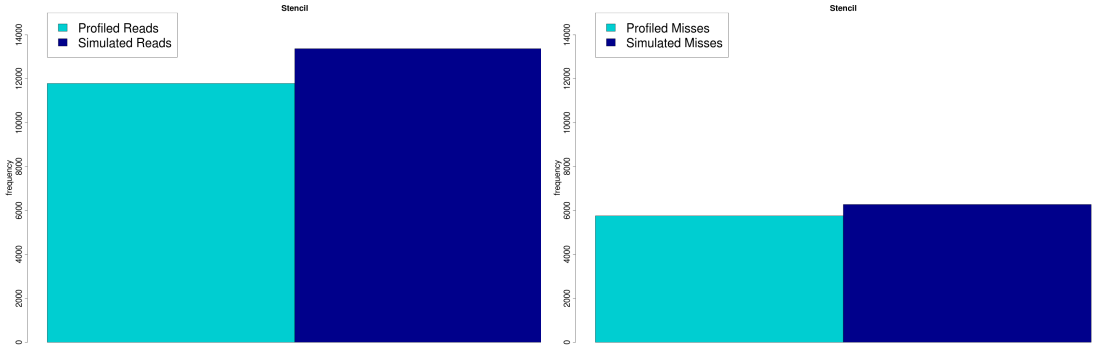
Relative standard deviation for our simulations is also under 2%, since with so many workgroups the chance of data reuse from randomly choosing adjacent workgroups is diminished. However there is variance in number of simulated reads which does not occur in the other two benchmarks. This is due to edge values in the stencil not being updated, so some threads aren't present to make up a full warp size. Therefore if workgroups containing these warps are randomly chosen to be simulated then they will be less reads.



(a) Reads (b) Misses

Figure 8.11: Relative standard deviation of results for 128 x 128 x 32 stencil benchmark

Our simulation results for reads and misses can be seen in figure 8.12 where we overestimate the number of reads by 13.5% to 13368. However the number of misses are slightly closer to profiled but still higher with a percentage error of 9%. This culminates in the simulated miss rate being 2% lower at 46.9%, rather than the profiled 48.8% as shown in figure 8.13, which is accurate given the size of the benchmark.



(a) Reads (b) Misses

Figure 8.12: Number of reads and misses for the 128 x 128 x 32 stencil benchmark

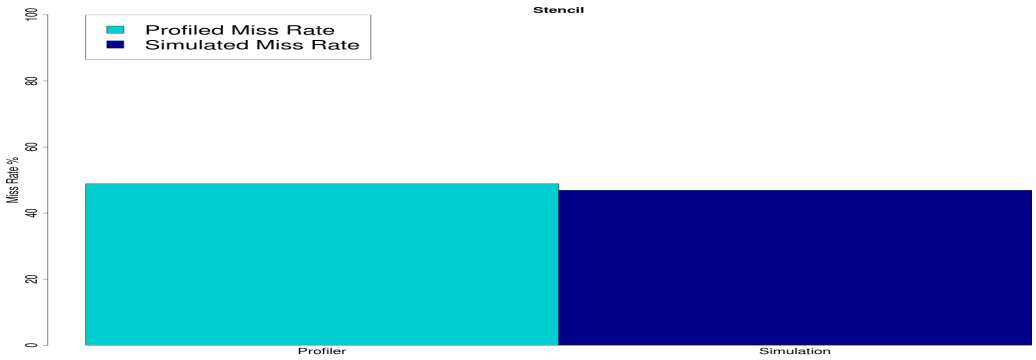


Figure 8.13: Miss rate for the 128 x 128 x 32 stencil benchmark

8.4 Summary

In this chapter we begin with our results from the matrix transposition benchmark which we can predict the miss rate for exactly. This is what we would expect since for matrix transposition's access pattern the miss rate is always 100%.

The matrix multiplication benchmark is more complex but we can still accurately predict the number of reads for all configurations. Our estimation of misses is also accurate to a percentage error of 19% up until the number of workgroups exceeds 60. As this is the workgroup capacity of a single SM, after which the extra workgroups are stalled. A feature which is not accounted for and results in an underestimation of the number of misses. As a result we predict the miss rate to around 6% which is accurate for benchmarks with less than 60 workgroups, but once we exceed this boundary our profiled miss rate jumps to 12% which we don't predict.

The stencil benchmark was only experimented with using a single configuration due to availability of binary input files needed to populate the data. This gave us a large size of 476,280 threads in 7560 workgroups. Due to 7560 being perfectly divisible by the number of cores on our machine however we could still achieve good results. Therefore we estimate the miss rate to be 46.9%, close to the 48.8% profiled.

Chapter 9

Conclusion

In this dissertation we have shown that by taking memory access traces from OpenCL kernel execution on a non-specific device we can use a simulation to recreate the kernel's cache performance on a NVIDIA Fermi GPU. In this chapter we evaluate our contributions to the project as well as the results obtained. Scope for future work in the field is also described as well as improvements to the existing simulation.

9.1 Summary

In chapter 4 the method used to obtain traces of OpenCL execution is described. This involves using the LLVM compiler infrastructure to instrument the intermediate representation of the OpenCL kernel we are interested in. Two instrumentations are then done, the first to find the length of the trace so that memory can be optimally allocated. While the second records for each memory access not only the address accessed and whether it was a read or a write, but also metadata to facilitate scheduling like the thread which made the access.

Once we have this access trace in a machine dependent sequence, we need to rearrange it so it represents an order similar to Fermi using the scheduler discussed in chapter 5. Fermi GPUs execute instructions in a SIMT pattern, with a 32 thread warp accessing a single instruction. Using information about any loops each instruction was in when executed we can successfully match memory accesses from the same SIMT execution together. Other more simplistic algorithms are also implemented for visual comparison when plotting the scheduled memory trace as a graph of execution order against data element referenced.

To then estimate the cache performance of this memory trace we process it using a L1 cache simulator described in chapter 6. Our simulator aims to recreate the coalescing of memory accesses from the same warp as occurs in hardware for a L1 cache on a SM. In order to do this not all the access are simulated but only those from selected workgroups that are assigned to that SM.

The performance statistics from our cache simulation are then compared against the profiled cache counters from the same OpenCL kernel on an Fermi GTX480 to give the results in chapter 8. These results show that if the number of workgroups doesn't exceed 60 then we can accurately predict the number of misses to within 20% error in the worst case, after this point then extra workgroups are stalled and assigned dynamically to cores which is not simulated for. Overall we have met our initial objective as for all the benchmarks the miss rate simulated doesn't differ from the profiled miss rate by more than 6%, allowing for a reasonable estimate of kernel GPU cache performance.

9.2 Future Work

Future work on this project could involve extending the simulation to incorporate the dynamic scheduling of stalled workgroups once the number of workgroups per SM exceeds capacity, as this is the main source of error in our results. This would involve extending the cache simulator so that it is aware of the machine dependent limitations of maximum number of warps and workgroups per SM. Our simulation should then stall the extra work if these boundaries are reached before allocating the work when space becomes free.

More through validation could also be done using other Fermi architectures apart from the GTX480 with different compute capabilities. This would ensure that machine dependent factors like number of cores and maximum workgroups per core are properly accounted for. As well as this a full benchmark suite could be converted so that all the benchmarks use our wrapper, since this would provide a more varied set of kernels to validate with. Testing both the accuracy of our simulation and the flexibility of our instrumentation.

Additionally if the simulation could then be successfully validated against a whole benchmark suite this would provide a platform for experimentation to be done into the effects of OpenCL parameters. This could include investigation into the effects of increasing warp size on performance, as well as the impact of workgroup size. Further work could also be done to try to simulate large warp formation and two-level warp scheduling as proposed by Narasiman et al [13] to see how cache efficient they would be.

Investigation could also be done into the effects of different cache parameters on kernel performance. As our cache simulator is configurable to a range of associativities and sizes, so these experiments could be easily automated. Therefore for different types of OpenCL kernel the optimal cache configuration could be explored. Going even further the cache simulator could be expanded to take into account both L1 and L2 layers of the cache. Although this presents challenges as all the L1 caches would need to be simulated to ensure that the L2 responds to all the evicted cache lines.

9.3 Critical Evaluation

Ideally our simulation would be validated against more benchmarks than the three used, as these were specifically selected for their easy to understand access patterns and large amounts of data reuse. As such they are not representative of a full benchmark suite. Most glaringly none of the kernels validated had any branching which has a major impact of coalescing and parallel performance, and so would more rigorously test our simulation. The majority of practical real world programs have branching so this also limits the usability of our simulation. None of the kernels we used have any barriers or memory fences either, another common feature of real world programs.

In retrospect it would also have been more effective to perform coalescing in the scheduler rather than cache simulation, as additional data is available to more accurately make the decision. This would also condense the number of trace entries making the scheduler faster. As currently the run time is up to 45 minutes for the stencil benchmark containing 3.8 million trace accesses, violating the motivating factor of a lightweight simulation. The time complexity could have been further reduced with more thought into the central structure of the scheduler beforehand, as doubly linked lists would have allowed for in-place rearrangement with less time spent reallocating memory.

However using traces as simulation input works well by being a versatile strategy due to the range of kernels that could potentially be traced. Traces are extensible as well thanks to the ability to gather other information if it is ever required. A good example of which is finding loop data for solving the challenge of reconstructing warps from accesses inside loops. Visualization also complements tracing well by conveying which accesses are from of each thread, especially useful for large traces. Where plotting allows the different scheduling policies and kernel access patterns to be clearly conveyed.

Bibliography

[1] Wittenbrink C.M. ; Kilgariff E. ; Prabhu A. Fermi GF100 GPU architecture. *Micro, IEEE (Volume:31, Issue 2)*, March 2011.

[2] TM Aamodt and WL Fung. GPGPU-sim 3.x manual, 2012.

[3] AMD. *AMD APP SDK Getting Started Guide v2.8*.

[4] Wilson WL Fung and Tor M Aamodt. Thread block compaction for efficient SIMT control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 25–36. IEEE, 2011.

[5] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007.

[6] Intel. *T2390 Pentium datasheet* `http://ark.intel.com/products/35153/Intel-Pentium-Processor-T2390-1M-Cache-1\_86-GHz-533-MHz-FSB`.

[7] Khronos. *OpenCL 2.0 Reference Pages*.

[8] Erik H.D'Hollander Kristof Beyls. Reuse distance as a metric for cache behavior. *IN PROCEEDINGS OF THE IASTED CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS*, 2001.

[9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[10] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Fermi architecture white paper. 2009.

[11] LLVM. *Analysis and Transform Passes document - mem2reg section*.

[12] LLVM. *Command guide 3.4, optimizer section*.

[13] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.

[14] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. A detailed GPU cache model based on reuse distance theory. *Eindhoven Universtiy of Technology*, 2013.

[15] NVIDIA. *Compute Command Line Profiler User guide*.

[16] NVIDIA. *CUDA Toolkit v5.5 Release Notes*.

[17] NVIDIA. *Geforce 256, www.nvidia.com/page/geforce256.html*.

[18] NVIDIA. *GTX480 Web Datasheet,www.nvidia.co.uk/docs/IO/90190/GTX-480-470-Web-Datasheet-Final4.pdf*.

[19] NVIDIA. *NVIDIA Developer Zone, CUDA GPUs,* `https://developer.nvidia.com/cuda-gpus`.

[20] NVIDIA. *Profiler users guide, http://docs.nvidia.com/cuda/profiler-users-guide/*.

[21] CUDA Nvidia. Programming guide, 2008.

[22] CUDA NVIDIA. GPU occupancy calculator. *CUDA SDK*, 2010.

[23] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. `http://www.cs.ucla.edu/~pouchet/software/polybench`, 2012.

[24] Simon Moll. *Axtor source code,* `https://bitbucket.org/gnarf/axtor/`, 17.09.12.

[25] Alan Jay Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *Software Engineering, IEEE Transactions on*, pages 121–130, 1978.

[26] Alan Jay Smith. Cache evaluation and the impact of workload choice. *12th Annual International Symposium on Computer Architecture*, June 1985.

[27] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[28] Tao Tang, Xuejun Yang, and Yisong Lin. Cache miss analysis for gpu programs based on stack distance profile. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 623–634. IEEE, 2011.

[29] University of British Columbia. *CPSC 313: Computer Hardware and Operating Systems,Winter 2012, Assignment 4*.

[30] William N Venables, David M Smith, R Development Core Team, et al. An introduction to r, 2002.

[31] Jingling Xue and Xavier Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *Computers, IEEE Transactions on*, 53(5):547–566, 2004.