A Detailed GPU Cache Model Based on Reuse Distance Theory

Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal
Eindhoven University of Technology
{c.nugteren, g.j.w.v.d.braak, h.corporaal}@tue.nl

Henri Bal
Vrije Universiteit Amsterdam
bal@cs.vu.nl

Abstract

As modern GPUs rely partly on their on-chip memories to counter the imminent off-chip memory wall, the efficient use of their caches has become important for performance and energy. However, optimising cache locality systematically requires insight into and prediction of cache behaviour. On sequential processors, stack distance or reuse distance theory is a well-known means to model cache behaviour. However, it is not straightforward to apply this theory to GPUs, mainly because of the parallel execution model and fine-grained multi-threading. This work extends reuse distance to GPUs by modelling: 1) the GPU's hierarchy of threads, warps, threadblocks, and sets of active threads, 2) conditional and non-uniform latencies, 3) cache associativity, 4) miss-status holding-registers, and 5) warp divergence. We implement the model in C++ and extend the Ocelot GPU emulator to extract lists of memory addresses. We compare our model with measured cache miss rates for the Parboil and PolyBench/GPU benchmark suites, showing a mean absolute error of 6% and 8% for two cache configurations. We show that our model is faster and even more accurate compared to the GPGPU-Sim simulator.

1. Introduction

In the past decade, graphics processing units (GPUs) have emerged as a popular platform for non-graphics computations. Through languages such as OpenCL and CUDA, programmers can use these massively parallel architectures for domains such as linear algebra, image processing and molecular science. To counter the imminent memory wall [9], GPUs have been equipped with software-managed (scratch-pad) and hardware-managed (cache) on-chip memories. In particular for integrated solutions with general-purpose memories (e.g. ARM Mali, Xbox One) off-chip memory bandwidth is scarce: using the on-chip memories efficiently is paramount to exploit the GPU's full potential.

Because GPUs are designed to hide their memory latencies through fine-grained multi-threading, the goal of a GPU's on-chip memory is not to reduce latencies as is the case for CPUs. Instead, the GPU's on-chip memories serve the purpose of reducing the off-chip memory traffic. An increased cache hit rate will translate to performance improvements for memory-intensive programs, as off-chip memory traffic (the performance limiting factor) is decreased proportionally. In fact, many GPU programs are memory bandwidth intensive: for an example set of benchmarks, this is as much as 18 out of 31 [13]. Specific examples of cache optimisations include cache blocking for sparse matrix vector multiplication (5x speed-up) [24] and tiling for a stencil computation (3x speed-up) [19].

Since GPUs rely on their on-chip memories to reduce off-chip memory traffic, optimising GPU programs for cache locality has become important for performance and energy. However, to be able to perform cache locality optimisations efficiently, *insight* into the types of cache misses and a *prediction* of the amount and source of cache misses is essential, as shown for example in [4, 12, 16]. A cache model can also be used to guide compilers to select their optimisation parameters, e.g. a loop-tiling factor and a thread coarsening factor. An example is the polyhedral model based C-to-CUDA compiler PPCG [22], which leaves the problem of tile-size selection to the programmer because of a lack of insight into cache behaviour. Additionally, a model can accelerate design space exploration, i.e. finding cost-efficient values for cache parameters such as associativity or the cache-line size. An analytical cache model can thus help to obtain insight into cache usage, to guide programmers and compilers, and to evaluate the effects of cache parameters on cache miss rates.

A well-known cache model is the 3C model [10], distinguishing three types of cache misses: 1) compulsory (or cold): misses because of a first time access, 2) capacity: misses because of a limited cache size, and 3) conflict: misses due to a limited cache associativity or a non-ideal replacement policy. To estimate the amount of cache misses based on the 3C model, a *reuse distance* profile (or 'stack') can be constructed from a memory access trace [4]. The reuse distance theory keeps track of memory requests, moving recently used addresses to the top of an address stack. Addresses not yet present in the stack are the compulsory misses, and addresses with a stack depth larger than the cache size are the capacity misses. Although this model does not take conflict misses into account, it gives a good lower bound for the total miss rate on sequential architectures [4] and even on multi-core CPUs [18].

Existing performance and power models for GPUs (e.g. [2, 11]) have not included a cache model up to

now: they are only valid for (older) GPUs without data caches. However, understanding cache behaviour is important as off-chip memory bandwidth is becoming increasingly scarce relative to compute power [9]. The main challenges of creating a cache model for GPUs lie in the execution model: as we will see in this paper, fine-grained multi-threading and parallelism make it non-trivial to find the order in which memory requests appear to the cache. Because reuse distance theory can only be applied to an ordered memory access trace, it is not directly suited for GPUs. This work extends the reuse distance theory to model GPU caches through the following extensions:

1. The reuse distance theory is adjusted to match the GPU's **parallel execution** model. This includes modelling threads, warps, threadblocks, cores, and sets of active threadblocks.

2. The GPU's **memory latency** is modelled by keeping track of in-flight accesses and their arrival times, introducing a new type of misses: *latency misses*. Furthermore, the memory's non-uniformity is modelled by sampling from a half-normal distribution.

3. Limited **associativity** is modelled by creating a private reuse distance stack per cache set. We identify the mapping of addresses to sets with micro-benchmarks.

4. The effects of **miss-status holding-registers** (MSHRs) are modelled, which store in-flight memory request information.

5. Threads within a warp are executed in lock-step, but individual warps can make different progress. This **warp divergence** is modelled by simulating a thread-pool from which warps can be selected for execution.

The model is implemented in C++ (optimised for performance) and the source-code is available on-line[1], including a custom CUDA memory access tracer for the Ocelot emulator [7]. Our contributions can be summarised as follows:

- Five extensions to the reuse distance theory are proposed, creating a detailed cache model for GPUs (section 4). The model is validated for two cache configurations and for two benchmark suites (section 6).

- Two architectural details are found through micro-benchmarking: 1) the GPU's mapping of addresses to sets, and 2) the number of MSHRs (section 5).

- The usability of the model is demonstrated by showing an example cache parameter sweep (section 7).

[1] http://github.com/cnugteren/gpu-cache-model

This work focuses on the GPU's L1 data caches: after it is known in what order memory accesses appear in the L1 cache and which of those miss, existing multi-core CPU models can be applied to model the GPU's L2 cache.

2. Related work

There is only a single other complete GPU cache model presented in the literature (to the best of our knowledge). This model by Tang et al. [21] is also based on reuse distance theory. However, there are a number of reasons why we propose a new cache model. First, in contrast to our work, Tang et al. model only a single threadblock, assume warps to execute in lock-step, do not model MSHRs and the mapping of addresses to sets, and do not give any details on the used memory latency model. Second, their validation is very limited: 1) they validate against a GPU simulator, not against real hardware, and 2) they include only basic, hand-picked kernels with non-representative input data-sizes. Third, their model is limited to kernels that can be statically analysed. This is in contrast to our approach, where we support any GPU kernel: we use an emulator to generate traces. Our final reason is practical: their model is not available in the form of source-code or binary.

Another cache model [15] is part of a complete GPU model, but assumes hit and miss rates to be known. Furthermore, other work has used reuse distance to analyse non-GPU multi-core and many-core workloads [6, 17, 18]. In contrast to our work, they investigate cache contention caused by running multiple programs on different cores. Because they do not target GPUs, many of their assumptions (e.g. no data reuse among threads, execution order known) are not valid for our work (and vice versa).

3. Background

This section briefly introduces the GPU execution model, the cache architecture, and the reuse distance theory. Additional background information on GPUs can be found in the CUDA programming guide [14] and on caches and reuse distance theory in literature [5].

We use NVIDIA's Fermi architecture as an example throughout this paper and experiment on a GeForce GTX470 GPU (but can be applied to others as well). The Fermi architecture has up to 16 cores (also known as *streaming multiprocessors* or *compute units*), of which 14 are available in the GTX470. The cores each contain 32 processing elements and share a 64KB on-chip data memory, configurable as a combination of a scratchpad and a L1 cache (16/48KB or 48/16KB). All cores share a larger L2 cache (up to 768KB). This work focuses on the L1 data cache, the main challenge of modelling GPU caches. The GPU's L1 cache handles only off-chip loads: stores are handled by the L2 cache only, not by the L1 cache [14]. Therefore, only loads are considered in this work, although the presented cache model can be applied to stores as well.

The cache-related terminology used is as follows [5]. 'Cache-line' describes a location in the cache, while 'cache-block' refers to the data that goes into a cache-line. Furthermore, $\mathbf{S}$ represents the number of sets in a cache.

3.1. The CUDA/OpenCL execution model

The programming frameworks CUDA and OpenCL allow programmers to specify small programs (*kernels*) that are executed multiple times. Each instance of a kernel (*threads* in CUDA terminology, *workitems* in OpenCL terminology) has its own unique identifier in order to work on different data. Programmers furthermore divide all their threads in fixed-sized blocks (*threadblocks* in CUDA, *workgroups* in OpenCL). Within a threadblock, threads share an on-chip local memory and can synchronise through barriers. However, there is no synchronisation or communication support among threads in different blocks.

In a Fermi GPU, a threadblock is mapped in its entirety onto a core. Together, threads from one or more threadblocks can form a set of *active* threads on a single core. For Fermi GPUs, this is limited to 8 threadblocks or 1536 threads, whichever limit is reached first [14]. Such a set of active threads executes concurrently in a multi-threaded fashion as *warps* (NVIDIA terminology) or *wavefronts* (AMD terminology). In the Fermi architecture, a warp is a group of 32 threads executing in lock-step in an SIMD-like fashion on a single core, dividing the workload over the core's processing elements [14].

3.2. The GPU cache architecture

The Fermi GPU has multiple data caches: a L1 cache for each core and a shared L2 cache. Fermi has a 16KB 4-way associative L1 cache, which can store 128 *cache-lines* of 128 bytes each [23]. The 128 lines are divided over 32 sets, each containing 4 lines: every memory address is mapped onto one of the sets of the cache using a *mapping function*. Reads to the off-chip memory are cached in L1, writes are not. The GPU's cache replacement policy is unknown, however, the simulator GPGPU-Sim [3] assumes a least-recently-used (LRU) policy, although no proof is given.

3.3. Reuse distance theory

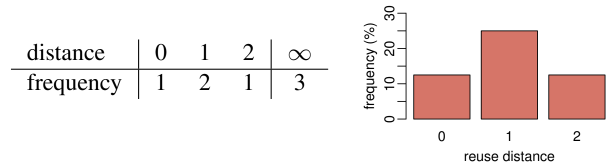
Given an ordered memory access trace, a reuse distance profile (or stack) [4] can be computed as follows. For each access, the reuse distance is the number of unique addresses accessed between this access and the most recent previous access to the same address. When there is no previous access, the distance is set to infinity (∞). Constructing a reuse distance profile can be done at for example address granularity or at cache-line granularity. An example of both is given in table 1, assuming a cache-line size of 4 elements (time progresses from left to right).

A reuse distance profile can be used directly to obtain cache hit/miss rates. Given a fully-associative cache of n lines with a least recently used (LRU) replacement policy,

Table 1. Reuse distance example

access	x[0]	x[5]	x[3]	x[9]	x[3]	x[3]	x[5]
address	0	5	3	9	3	3	5
distance	∞	∞	∞	∞	1	0	2
cache-line	0	1	0	2	0	0	1
distance	∞	∞	1	∞	1	0	2

any access with a reuse distance d larger than or equal to n ($d \geq n$) will miss. Vice versa, when $d < n$, the access will hit in the cache. In this way, the reuse distance profile at cache-line granularity gives the compulsory miss rate ($d = \infty$) and the capacity miss rate ($d \geq n$ and $d \neq \infty$). For our example in table 1, given a cache size of 2 lines, we find 3 compulsory misses (42%), and 1 capacity miss (14%). A reuse distance profile can also be visualised by constructing a histogram, containing all necessary data to compute compulsory and capacity miss rates. A histogram for our example data from table 1 is given in figure 1 (at cache-line granularity).



distance	0	1	2	∞
frequency	1	2	1	3

Figure 1. A table and histogram reuse distance profile for the example from table 1

4. Reuse distance for GPUs

Reuse distance theory can only be applied to an ordered memory access trace[2]: finding this order for a GPU is not trivial. This section discusses five extensions to the reuse distance theory, four of which are related to finding the order in which memory accesses appear to the cache. The theory is extended by: 1) integrating the GPU's parallel execution model of threads, warps, threadblocks and sets of active threads, 2) introducing non-uniform memory latencies, 3) modelling cache associativity, 4) modelling MSHRs, and 5) modelling warp divergence. Furthermore, implementation details of the model are given.

4.1. The GPU's parallel execution model

A GPU typically executes thousands of small, light-weight threads. Because of the parallelism expressed in the execution model, these threads can to some extent be executed independently on different cores. Furthermore, due to limited resources (e.g. register file, scratchpad memory), not all threads can be *active* at the same time, i.e. eligible for execution. Tang et al. [21] argue that there is limited reuse across different threadblocks on the same core: they

[2]The traces used in this work are not obtained from simulation: they are rather unordered lists of memory accesses and only contain ordering information with respect to a single thread.

model only a single block of threads on a single core. To create a more realistic model, we do model complete sets of active threads (one or more threadblocks). Furthermore, we model multiple of such sets and multiple cores (because the workload can vary for different cores).

To determine which threads execute together as a set of active threads on a single core, we follow Fermi's execution model (an example is shown in figure 2). First, threadblocks are divided round-robin over the cores until they are full. Then, a new threadblock is scheduled when another threadblock is done (first-done, first-serve). For each core, threadblocks are grouped in sets of active threads according to the block-size and the resource limitations as listed in section G.1 of the CUDA programming guide [14]. Furthermore, threads in a warp are scheduled simultaneously. Determining the scheduling *order* among warps in a set of active threads is not straightforward (e.g. dependent on thread workload and cache contents): this is approximated step-by-step in the remainder of this paper.

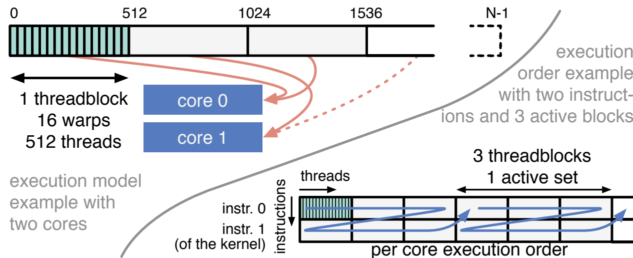


Figure 2. Execution model examples

Transforming the parallel execution model into an ordered memory access trace can be done by: 1) applying the GPU's thread-scheduling policy, and 2) by taking into account pipeline and memory latencies. For now, we assume a basic round-robin scheduling policy among warps in a set of active threads (divergence is discussed in sections 4.4 and 4.5), and zero-latency hardware (latencies are discussed in section 4.2). Now, for a given kernel, its execution can be sequentialised to obtain an instruction trace. We illustrate this with an educational example: a kernel with 4 threads, each performing 2 loads (`x[2*tid]` and `x[2*tid+1]`, for which '`tid`' denotes a thread's unique identifier). Given the round-robin scheduling of threads and no latencies, we obtain the reuse distances (for lines) as shown in table 2, assuming a cache-line size of 4 elements, a single thread per warp, and only a single set of threads on a single core.

Table 2. GPU reuse distance computation

instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
distance	∞	0	∞	0	1	0	1	0

However, before a reuse distance profile can be constructed from a given thread order, memory requests need to be combined according to the GPU's *memory coalescing* capabilities. Coalescing is applied in specific cases, for example when threads from a single warp access the same cache-line. Coalescing is implemented according to the specifications of the GPU architecture, as described in section G.4.2 of the CUDA programming guide [14].

4.2. Memory latencies

In reuse distance theory for sequential processors it is assumed that either: 1) memory latencies are non-existent, or 2) memory accesses cannot overtake each other. Although individual threads on a GPU execute in-order, these assumptions are not valid across different GPU threads. Moreover, the GPU's memory latencies are typically high compared to CPU latencies. Therefore, the reuse distance theory is extended to model the GPU's latencies.

First of all, the notion of time is introduced. Every column in the reuse distance theory is assigned with a monotonously increasing time-stamp (not reflecting actual processor cycles or time). Now, each access is assigned a specific latency to delay its effect. We illustrate this based on the same example as shown in table 2 with 2 accesses and 4 threads. Table 3 shows the updated results: every memory request occurs at a fixed time (0–7) and is assigned a latency (a fixed value of 2 time units in this example). Accumulation of an access's latency with its issued time-stamp determines when the request will have effect in the cache. We show this in the '*effect at*' row of table 3. Now, computation of the reuse distances is no longer based on the '*cache-line*' row, but on the new '*cache effect*' row, as shown in the table. In this particular example, the cache-line data is simply shifted by 2 time-stamps (highlighted).

Table 3. Reuse distance with fixed latencies

time	0	1	2	3	4	5	6	7	8	9
instruction	0	0	0	0	1	1	1	1	-	-
thread ID	0	1	2	3	0	1	2	3	-	-
address	0	2	4	6	1	3	5	7	-	-
cache-line	0	0	1	1	0	0	1	1	-	-
cache effect	-	-	0	0	1	1	0	0	1	1
distance	∞	∞	∞	∞	0	1	0	1	-	-
hit/miss	m	m	m	m	h	h	h	h	-	-
latency	2	2	2	2	2	2	2	2	-	-
effect at	2	3	4	5	6	7	8	9	-	-

Adding the notion of time and latency changes the reuse distances obtained. This can be seen for example by comparing the distances found in tables 2 and 3. The addition of latencies can thus transform capacity misses in hits and vice-versa. However, this approach can also introduce additional infinite distances (∞) that are not compulsory misses. To repair this, the notion of *latency misses* is introduced:

requests that miss in the cache because an earlier request to the same cache-line is still in-flight.

So far, we have modelled only a fixed latency. To better reflect the reality, two additional aspects are also modelled: 1) conditional latencies applied depending on the reuse distance, and 2) non-uniform memory latencies. In this case, we need to distinguish between cache hits (to model the pipeline latency) and cache misses (to model the memory latency). This requires us to embed information about the cache size in the model, making the reuse distance profile no longer cache-size independent.

The example of table 3 is extended to include a hit latency of 0 and a miss latency of 2. If we furthermore assume a cache-size of 2 lines, the results as shown in table 4 are obtained. We observe that the reuse distances change again, influenced by the reduced latency of the last 4 memory accesses. Furthermore, multiple 'cache effects' can now occur simultaneously at a single time-stamp (highlighted in the table). Such simultaneous accesses are handled in the order in which the memory accesses were issued.

Table 4. Extended with conditional latencies

time	0	1	2	3	4	5	6	7
instruction	0	0	0	0	1	1	1	1
thread ID	0	1	2	3	0	1	2	3
address	0	2	4	6	1	3	5	7
cache-line	0	0	1	1	0	0	1	1
cache effect	-	-	0	0	1 0	1 0	1	1
distance	∞	∞	∞	∞	0	0	1	0
hit/miss	m	m	m	m	h	h	h	h
latency	2	2	2	2	0	0	0	0
effect at	2	3	4	5	4	5	6	7

This theory is extended to a more realistic model by clipping the 'effect at' time to the time of a still in-flight request for the same cache-line (if present). This will for example change the 'effect at' time of the request at time-stamp 1 in table 4 from 3 to 2, as the request for cache-line 0 was already made at time-stamp 0.

Finally, the non-uniform latency of accessing the GPU's off-chip memory is modelled. Because a detailed model of the memory latency is beyond the scope of this paper (it requires a full GPU model or simulator, including e.g. the pipeline and interconnect), a probabilistic approach is taken. The memory latency is modelled as $\lambda_{min} + |\mathcal{N}(0, \sigma^2)|$: a fixed minimum latency λ_{min} offset by the absolute value of a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with zero mean, i.e. a half-normal distribution. The parameters to set are the memory's best-case latency λ_{min} and a measure for the memory's non-uniformity: the standard deviation σ of the half-normal distribution.

The 'latencies' discussed in this section are not real latencies: the cache model is not a complete GPU model and does not have a notion of actual clock cycles. For example, there can be a varying number of non-memory operations between two memory accesses, affecting latency greatly. To model the effects of non-memory operations would require integration with a complete GPU model, which is beyond the scope of this work. Therefore, the introduction of latencies to our model should be seen as a way to capture the global ordering roughly rather than as a way to obtain an exact reuse distance profile.

4.3. Cache associativity

The reuse distance theory models the compulsory and capacity misses, but does not take into account misses caused by the limited associativity of a cache (part of the conflict misses). It has been shown that such misses form a relatively small percentage of the total amount of misses for sequential processors, even in the case of a direct mapped cache [4]. However, typical GPU programs are more sensitive to associativity, because they often show regular memory access patterns on large data structures (e.g. matrix or image operations). To improve the accuracy, we extend the reuse distance theory to model cache associativity.

The reuse distance theory can be extended to model associativity as follows. Instead of keeping track of a single reuse stack, a private stack is created for each set in the cache. In that way, a set becomes a small cache with a size in lines equal to the number of ways, i.e. the associativity. For a fully-associative cache, this reduces again to a single stack because it has only a single set.

Along with the introduction of multiple sets (and their corresponding reuse stacks), we need to define a mapping of memory addresses to sets. Such a mapping can be either obtained directly by taking the last $log_2(\mathbf{S})$ bits from the line address, or by a more advanced *hashing function*, creating a hash-associative cache [5]. The simulator GPGPU-Sim [3] uses a direct mapping for Fermi GPUs, but does not claim that this is realistic. Therefore, because Fermi's mapping function is not public knowledge, a micro-benchmark was constructed to find the mapping. Therefore, because Fermi's mapping function is not public knowledge, a micro-benchmark was constructed (see section 5), finding a hashing function with a 5-bits XOR operation for a Fermi GPU.

4.4. Miss-status holding-registers

A GPU can have only a finite number of memory requests pending: pending requests are stored in miss-status holding registers (MSHRs), per-core registers that keep track of *in-flight* (in progress) memory requests. The reuse distance theory is extended to model such registers to improve the accuracy of the cache model. MSHRs are organised in such a way that each entry can service a unique cache-line request: requests to the same cache-line are merged into a single entry (up to a certain limit). A limited amount of registers limits the number of outstanding memory requests: either all MSHR entries are occupied when a

new cache-line is requested, or an MSHR entry corresponding to a specific cache-line is full. In either case, the active warp will be stalled because it cannot perform any more memory requests. While waiting for an entry to become free, the GPU processes warps that do not require MSHRs.

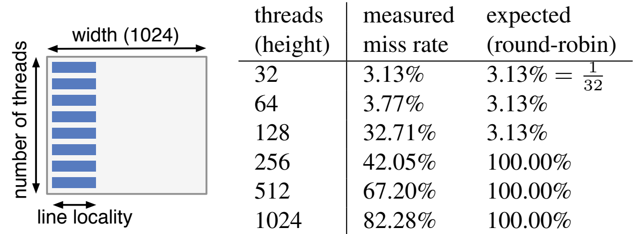
We model the limited amount of MSHRs, but assume that requests to the same cache-line are merged into a single entry. Our model keeps track of the number of unique outstanding memory requests. Before a warp modifies the reuse stack, it is ensured that it is either a hit or that the number of outstanding requests is not exceeding the number of MSHRs. If the warp cannot continue, it is put on-hold and issued again later. This is illustrated in table 5, in which the example of table 4 is shown, but now with the assumption that there is only a single MSHR available. Only threads 0 and 2 are shown to make the example concise. From table 5, we see that instruction 0 of thread 2 is cancelled and re-issued at a later time. Also, we see that instruction 1 of thread 2 (issued at time 4) does not have to be postponed: it uses the already occupied MSHR for cache-line 1.

Table 5. Extended with MSHR modelling

time	0	1	2	3	4	5	6
instruction	0	0	1	0	1	-	-
thread ID	0	2	0	2	2	-	-
address	0	4	1	4	5	-	-
cache-line	0	1	0	1	1	-	-
cache effect	-	-	0 0	-	-	1	1
distance	∞	∞	0	∞	∞	-	-
MSHRs used	0	1	0	0	1	-	-
status	miss	cancel	hit	miss	miss	-	-
MSHRs used	1	-	0	1	1	-	-
latency	2	-	0	2	2	-	-
effect at	2	-	2	5	6	-	-

Similar to the case of the hash function of the cache, it is not publicly known how many MSHRs a GPU core has. The GPU simulator GPGPU-Sim [3] uses a default of 32 MSHRs per core for a Fermi GPU, but does not claim that this value is realistic. Through micro-benchmarking (see section 5), we find that a Fermi GPU core has 64 MSHRs and a single warp can use only up to 6 MSHRs.

The relevance of modelling MSHRs is demonstrated with a simple experiment for the GPU's 16KB (128 lines) cache. The experiment consists of a kernel that performs a copy of a 2D matrix in a column-major fashion: each thread copies an entire row. Cache-line locality is not among threads (accesses are uncoalesced) but within each thread. Figure 3 illustrates the experiment and shows the results, varying the height of the 2D matrix from 32 to 1024 (equal to the number of threads: one threadblock only). A constant width of 1024 is set and a data-size of 4 bytes is used. The results show that the measured cache miss rates do not correspond to the miss rates when assuming an ordered round-robin schedule. We conclude from the table that, because of the limited number of MSHRs, certain threads run ahead of others. This can result in performance improvements (256–1024 threads) or losses (64–128) compared to a fair round-robin schedule. In other words: for a GPU cache model to be accurate, MSHRs need to be modelled.



threads (height)	measured miss rate	expected (round-robin)
32	3.13%	$3.13\% = \frac{1}{32}$
64	3.77%	3.13%
128	32.71%	3.13%
256	42.05%	100.00%
512	67.20%	100.00%
1024	82.28%	100.00%

Figure 3. The relevance of MSHR modelling

For example, when running 128 threads in round-robin, each cache-line can store a single cache-block (for a cache of 128 lines). However, when warps diverge, threads can run ahead and request new cache-blocks while others are still using their previous cache-block. Due to a non-oracle replacement policy, this can result in additional misses. On the other hand, when running 256 threads in round-robin, threads 128–255 overwrite the cache-blocks required by threads 0–127. In this case, divergence can only ameliorate cache behaviour: when threads run faster than others, they can benefit from their intra-thread cache-line locality.

4.5. Warp divergence

As a final extension, a warp divergence model is introduced. *Warp divergence* is defined as the process that causes program counters of warps to differ from each other as execution progresses. This is not to be confused with the non-cache related concept of *thread divergence*, which describes divergence within warps caused by branch instructions taken by a subset of threads in a warp. Instead, we discuss warp divergence: divergence among warps as a result of aspects such as on-chip local memory bank conflicts, non-uniform memory access latencies, instruction or data cache misses, and per-warp branches in program code.

Because we do not model the entire GPU and only have information on memory references, not all possible sources of warp divergence are modelled. Instead, the focus lies on the memory-related sources: 1) data cache hit and miss latencies, 2) non-uniform off-chip memory latencies, and 3) the limited number of MSHRs. The first two are introduced in section 4.2, and the third in section 4.4. This section models how these sources affect the warp execution order.

To model warp divergence, the concept of a warp queue is introduced. Initially, the queue is filled with all active warps (from one or more threadblocks) ordered by warp identifier (thread identifier modulo the warp size). As long as the queue is non-empty, a warp is selected based on a first-in first-out (FIFO) policy and a single memory request

is processed for each thread in the reuse distance model. After a warp finishes a memory request, it is not directly pushed to the back of the warp queue. Instead, it is delayed proportionally to the corresponding request's latency. Furthermore, if a warp does not succeed because all MSHRs are in use, it is sent to the back of the warp queue.

4.6. Implementation of the model

This section gives an overview of the implementation of the model and its infrastructure, as illustrated by figure 4.

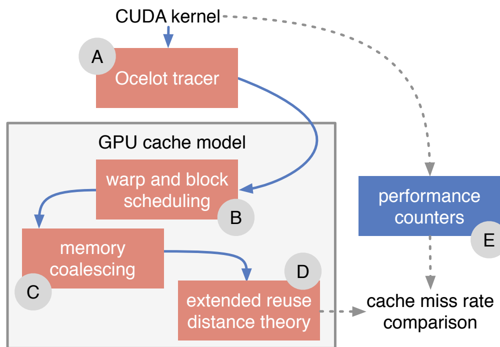


Figure 4. Infrastructure of the cache model

The Ocelot GPU emulator [7] is used to produce (unordered) memory access traces for CUDA kernels. A custom tracer (A) is implemented on top of Ocelot, creating a trace containing for each access: 1) the thread ID, 2) whether it is a read or a write, 3) the memory address, and 4) the size of the memory access. Because Ocelot does not simulate the GPU, the 'traces' are actually unordered lists of memory accesses rather than ordered traces that can be obtained from simulators. The only ordering in the traces is with respect to the instruction stream of a single thread.

Before the reuse distance theory can be applied, the memory accesses have to be ordered. Therefore, we first perform the allocation of threads to warps, warps to threadblocks, and threadblocks to cores (B). We follow the GPU's execution model as discussed in sections 3.1 and 4.1, and in section 4.1 of the CUDA programming guide [14]. The thread to warp allocation in particular can be modified for architecture exploration purposes, e.g. by changing the warp size or by implementing a strided assignment of threads to warps. Another possible modification is to incorporate dynamic warp scheduling to reduce warp branch divergence, performing for example thread block compaction or two-level warp scheduling. However, dynamic warp scheduling techniques might rely on details not available to the cache model, such as branch and warp divergence information. A solution for this is to implement such a dynamic warp scheduler in Ocelot (already used to produce traces). Rather than using this scheduler to change the schedule in Ocelot (which makes no sense for an emulator), it can be used to report a specialised thread to warp assignment that can be used within our cache model.

Next, a memory coalescing model (C) is implemented according to the behaviour as defined in section G.4.2 of the CUDA programming guide [14]. Coalescing is modelled before applying the reuse distance theory, as this can give a significant reduction in computational and memory complexity of the cache model: coalescing can compact the memory trace significantly.

All extensions to the reuse distance theory are implemented on top of the original theory (D). In this way, we can make use of the already available computational and memory efficient implementations for sequential processors [1]. A naïve implementation of a reuse distance stack has a computational complexity of $\mathcal{O}(NM)$, in which N is the trace length (the total number of memory accesses) and M the number of unique accesses. To handle the GPU's large number of threads and accesses, a more computationally efficient version is used: a binary-tree C++ implementation of Bennett and Kruskal's algorithm [1]. This implementation has a computational complexity of $\mathcal{O}(N \log(N))$, is independent of M, and gives a better scaling for traces where M is proportional to N. When modelling associativity, we increase the complexity by creating a binary-tree for each set in the cache. However, because the number of accesses per set is pre-computed, the size of each tree is reduced accordingly, achieving an overall comparable complexity. Further optimisations could be made to reduce the memory footprint (around 2GB for benchmarks from section 6), e.g. with a splay tree [1, 8].

To reduce the overall complexity and computational requirements, the number of threads can be limited in two ways: 1) a limited number of cores can be modelled, generalising results across all cores, and 2) a limited number of threads can be modelled. These core and thread counts are configurable parameters, set to a single core with up to 8192 threads for our experiments.

Finally, a verification method based on hardware counters (E) is included. NVIDIA's profiler NVPROF is used to output the measured number of cache-line hits and misses in the L1 data cache. The comparison of these numbers with the cache model's result is automated, producing the graphs as shown in the remainder of this work.

4.7. Overview of abstractions

The reuse distance theory assumes a least-recently-used (LRU) cache replacement policy, and so does our model. From the results of the micro-benchmark to find the associativity hash function (section 5.1, figure 5), we observe that the replacement policy resembles LRU in this case: oracle replacement would have caused a single miss only for each experiment. However, this is not a definite proof, and thus the commonly used LRU policy is assumed.

Synchronisation barriers at threadblock level are not included in the model. This could be added to the theory to

model warp divergence (and convergence) more precisely.

Different types of latencies are used in our model to represent in-flight memory requests and warp divergence. However, as discussed, this leaves the model in a grey area between a dedicated cache model and a full GPU model. To improve our latency and divergence model further, the model needs to be extended beyond caches only.

5. Micro-benchmarks

To complete the models of sections 4.3 (associativity) and 4.4 (MSHRs), additional information was obtained through micro-benchmarking: carefully designing a benchmark to extract details on the GPU architecture. This section describes these micro-benchmarks and the results.

5.1. Associativity micro-benchmark

The first micro-benchmark is designed to find the mapping of addresses to cache sets, crucial information to model associativity. Our micro-benchmark (shown in figure 5) launches a single block of 128 threads (4 warps), each performing 3 stages. In the first stage, each thread performs 32 coalesced loads designed to fill the entire 16KB of the L1 cache with subsequent addresses (an assumption at this point). This access pattern is repeated in the third stage while measuring the latencies of the individual loads. If we do not perform anything in the second stage, all loads show a low latency and are thus cache hits. This verifies our assumption. Now, performing a single load in the second stage will give increased memory latencies for some of the loads[3] in the third stage, as they become cache misses. By performing a sweep over different loads for the second stage, a mapping of addresses that belong to the same set is obtained. We find only up to 4 cache misses each time in the third stage as long as line-aligned accesses are performed: this is because of the 4-way associativity [23].

From the obtained mapping, the hashing function used to map addresses to sets is reverse-engineered. For the 16KB cache with 32 sets, we find that the 5 bits 7–11 and the 5 bits 13,14,15,17,19 of the byte-address are input to an XOR port to obtain a $log_2(\mathbf{S}_{16KB}) = 5$ bits set index, as shown in figure 6. The first gap in the address (the 12th bit) is a consequence of the cache configuration possibilities: Fermi's cache can also be configured as a 48KB 6-way associative cache with 64 sets. If the micro-benchmark is repeated, we find that the $log_2(\mathbf{S}_{48KB}) = 6$ set index bits are constructed by taking the 16KB's 5 bits (after the XOR operation) and prefixing bit 12.

To verify the found hashing function, an experiment with strided accesses is performed for the 16KB case. We construct a kernel with two identical loops, each time performing a number of non-overlapping 128-byte coalesced loads. The kernel is configured with a single warp only. The miss

[3]The number of misses is dependent on the order of accesses by the 128 threads and the cache replacement policy.

```
1  __global__
2  void mb1(int* mem, int* time, int sv) {
3
4    // Stage 1
5    for (i=0; i<32; i++)
6      temp = mem[tid + i*128];
7
8    // Stage 2
9    if (tid == 0)
10     temp = mem[sv];
11
12   // Stage 3
13   for (i=0; i<32; i++) {
14     start = clock();
15     temp = mem[tid + i*128];
16     time[tid + i*128] = clock() - start;
17   }
18 }
```

Figure 5. Associativity micro-benchmark (simplified code for illustration)

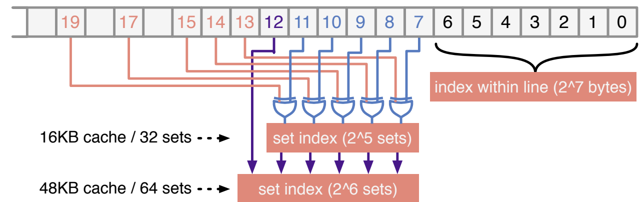


Figure 6. Usage of the byte-address bits

rate is measured at cache-line granularity using NVIDIA's profiler NVPROF. A sweep is performed over the number of loop iterations and the stride of the memory accesses. Figure 7 shows the results: either a cache miss rate of 100% (misses in both loops) or 50% (only misses in the first loop). The final row counts the number of set index bits varied across the loads, derived as the number of 50% miss rates in the row minus 1 (4 loop iterations always fit in a single set). The figure confirms the hypothesis, as the number of varied set bits (final row) corresponds to the number of bits included in the hashing function counting from the log_2 of the stride. For example, with a stride of 2^{12} and 128 loads, bits 12–18 are included, of which only 4 bits (13, 14, 15, 17) are used in the computation of the set index.

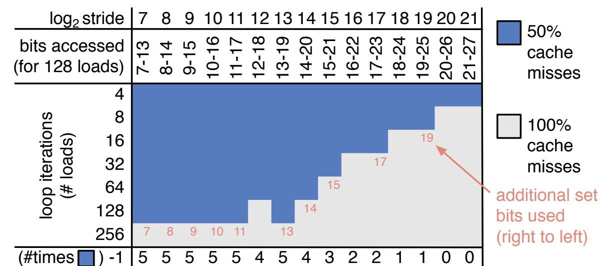


Figure 7. Hash function verification

5.2. MSHR micro-benchmark

Similar to the case of the hash function for associativity, it is not known how many MSHRs are available in the GPU. Therefore, we constructed the following micro-benchmark to find the number of MSHRs per GPU core. Initially, a CUDA kernel with only a single thread is launched. The kernel, as shown in figure 8, performs a configurable number of non-overlapping loads without dependences, which are timed in its entirety. The idea is that the GPU will issue multiple loads at a time, limited by the MSHRs. The results of this experiment are shown in figure 9 for a varying number of warps and a varying number of loads per warp (1 thread per warp as threads within a warp run in lock-step).

```
1  __global__ void mb2(int* mem, int* time) {
2    if (tid % 32 == 0) {
3      start = clock();
4
5      // Loop of independent loads (unrolled)
6      for (i=0; i<NUM_LOADS; i++)
7        temp = mem[32*(tid + i*NUM_WARPS*32)];
8
9      time[tid/32] = clock() - start;
10   }
11 }
```

Figure 8. MSHR micro-bench (simplified)

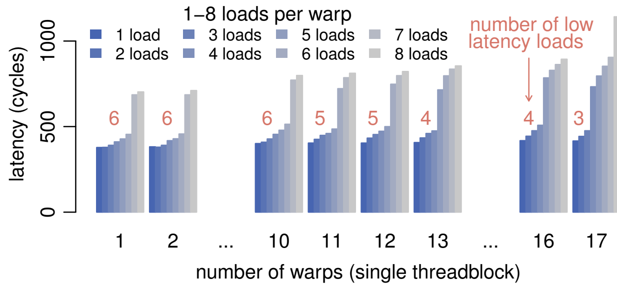


Figure 9. MSHR micro-benchmark results

When evaluating the results of figure 9 for a single warp (leftmost bars), we see that performing up to 6 loads yields a similar latency. When performing an additional 7th load, we observe a sudden increase in latency. From this data, we conclude that there are only 6 MSHRs available in this case: performing a 7th (or 13th, 19th, etc.) request increases the latency significantly. However, when evaluating the results of launching multiple warps, we observe that additional accesses can be performed without increasing the latency significantly[4]. In fact, this is true for up to 10 warps, allowing a total of $10 \cdot 6 = 60$ simultaneous requests. The figure shows a decrease to 5 loads per warp for 11 warps, 4 for 13 warps, and 3 for 17 warps. From this, we conclude that there are 64 MSHRs (e.g. 16 warps with 4 simultaneous requests each). We also conclude that a single warp is only allowed to use up to 6 entries, although this specific limit could be unrelated to the MSHR table, e.g. there could be a limit on the number of outstanding incomplete *instructions*.

6. Verification of the model

To demonstrate the usefulness and accuracy of the cache model, the modelled cache miss rates are compared against cache miss rates using hardware counters on a Fermi GPU and against a simulator. The verification is performed for both the 16KB 32-set 4-way and 48KB 64-set 6-way cache configuration on a GeForce GTX470 (newer Kepler GPUs also support 32KB [14]). To ensure a wide variety of GPU kernels, two complete benchmark suites are included: PolyBench/GPU[5] and Parboil [20]. The only exclusions made are the '*mb_sad_calc*' kernel from Parboil's '*sad*' benchmark, because it relies on the GPU's texture memory and texture cache, and the '*histo_main*' kernel from Parboil's '*histo*' benchmark, as it only uses atomic memory accesses. PolyBench/GPU is configured to use default data-sizes, and Parboil to use the 'medium' inputs (or 'large' where unavailable). For all Parboil benchmarks that run multiple iterations, the iteration limit is set to a maximum of 2. The two benchmark suites differ significantly: PolyBench/GPU contains mostly naive implementations of variants of matrix-multiplications (e.g. no on-chip local memory, limited parallelism), whereas Parboil contains optimised kernels of all sorts. Note that Parboil also contains benchmarks where 'caching' is performed manually in scratchpad memory. Their differences also become apparent by disabling the GPU's L1 data-cache in an experiment (the L2 is still enabled): the geometric mean performance drops by 5% (Parboil) and 15% (PolyBench/GPU).

6.1. Comparison against hardware counters

Using the infrastructure described in section 4.6, modelled and measured miss rates are collected for the two cache configurations. Figure 10 shows the results for the 16KB configuration, with kernel invocations on the x-axis. Bracketed letters are used in case a kernel is invoked multiple times. For each kernel, the left bar shows the modelled L1 data cache miss rate, and the right shows the measured miss rate using the profiler. The following types of measured misses are distinguished: 1) compulsory misses, 2) capacity misses, 3) associativity misses, 4) MSHR misses, and 5) latency misses. Latency misses are not included in figure 10's cache miss rate number: the profiler does not include these types of misses as they don't cause additional memory requests. We observe the following:

- The modelled compulsory misses (green) are lower or equal to the measured misses (blue) for all kernels.

[4] As the number of instructions increases when performing more loads or running more warps, the measured latency increases a bit as well.

[5] Available on-line at: `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`

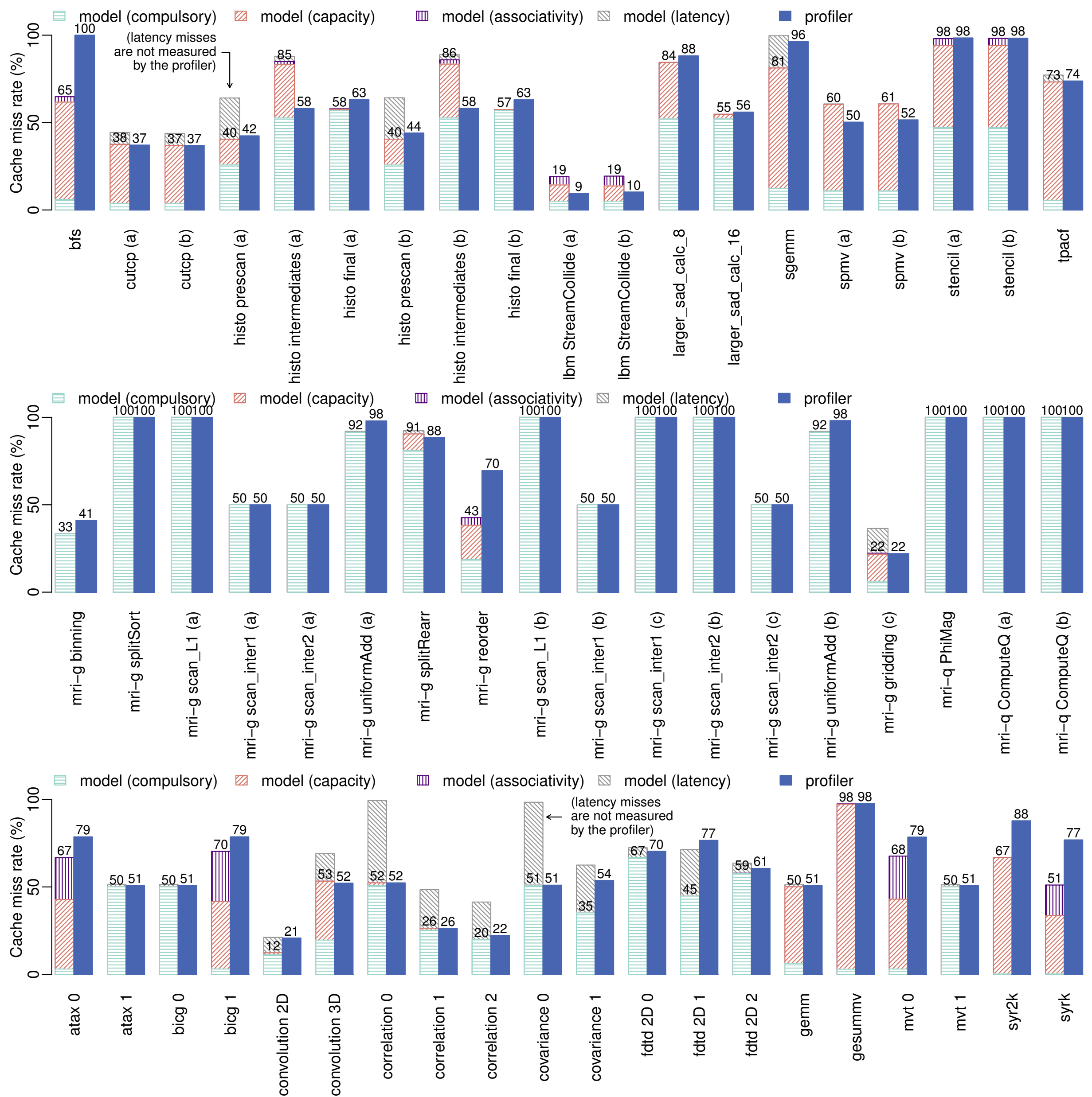


Figure 10. Results for Parboil (top and middle) and PolyBench/GPU (bottom), showing modelled (left) and measured (right) 16KB L1 miss rates: matching values represent a high modelling accuracy

This is important because the amount of compulsory misses is cache parameter independent. Furthermore, note that this results in a perfect model for cases where the only type of misses are compulsory, e.g. in many of the '*mri-g*' and '*mri-q*' kernels.

- Overall, most kernels show almost no associativity misses. However, there are still cases where associativity misses account for a significant fraction of the total amount of misses, in particular for the PolyBench/GPU benchmarks.

- These benchmarks show no additional misses caused by the limited number of MSHRs. In contrary, limiting the size of the MSHR table reduces the cache miss rate in many cases, as will also be shown in section 7.

- The kernels that show the largest difference between measured and modelled misses (e.g. '*bfs*', '*atax 0*', '*histo intermediates*') are very sensitive to the memory latency parameter. To improve the accuracy for these benchmarks, the model needs to be extended beyond caches only to obtain more realistic latency values.

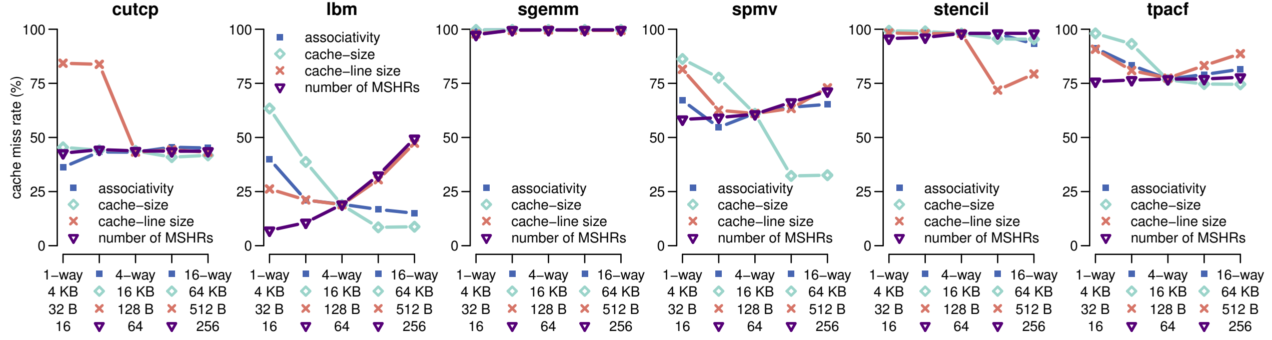


Figure 12. Evaluation of different values (x-axis) for four parameters (coloured series) for 6 kernels.

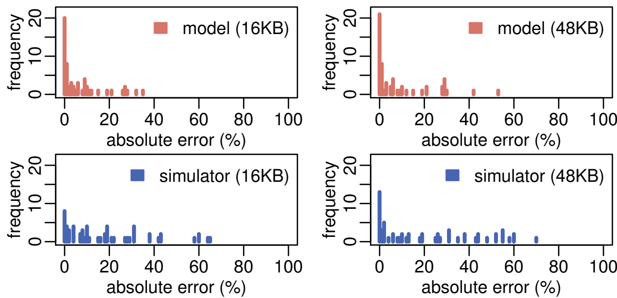


Figure 11. Absolute errors for the cache model (top) and for GPGPU-Sim (bottom)

The results of figure 10 are summarised in the top half of figure 11, augmented with the results for the 48KB configuration (not shown in detail). The arithmetic mean in absolute error[6] for our model is 6.4% for the 16KB configuration and 8.3% for the 48KB configuration. Finally, three scenarios are tested where a single component of the model is disabled each time, showing how much the introduced extensions to the reuse distance theory contribute to the precision of the model. The 6.4% arithmetic mean in absolute error changes as follows for the 16KB cache configuration: 1) a 9.6% error when associativity is not modelled, 2) a 12.1% error when latencies are not modelled, and 3) a 7.1% error when the number of MSHRs is unlimited.

6.2. Comparison against simulation

As a secondary verification metric, our cache model is compared against version 3.2.0 of the GPGPU-Sim simulator [3]. The (Fermi) simulator is configured with the specifications of the GTX470 GPU (both 16KB and 48KB caches) and runs the two benchmark suites: Parboil and PolyBench/GPU. The results are reported in the bottom half of figure 11, in which we show the absolute difference in cache miss rate compared to the results of the profiler. The simulator shows on average a larger error compared to our model: it produces a mean absolute error of 18.1% for the 16KB configuration and 21.4% for the 48KB configuration. Additionally, the run-time of the simulator is on average a factor 268x higher than our model. For example, GPGPU-Sim completes '*cutcp*' in 10 hours, whereas the model takes 10 seconds (excluding 4 minutes emulation in Ocelot).

[6]Note: the absolute error of a metric measured in percentages (miss rate) is also given in percentages.

7. Example use: evaluating cache parameters

To demonstrate the use of the model, a sweep over the cache parameters is performed. Evaluating all design points or finding optimal design points is beyond the scope of this work. Four different values are evaluated for the main parameters: 1) associativity, 2) cache-size, 3) cache-line size, and 4) the number of MSHRs. The values evaluated are 0.25x, 0.5x, 2x, and 4x the GPU's original value for the 16KB configuration. The results (figure 12) include 6 benchmarks from Parboil, chosen because of their mix of different types of cache misses. We observe the following:

- Associativity is a parameter of little importance for the evaluated benchmarks. Small benefits of a high associativity are only visible for '*stencil*' and '*lbm*', benchmarks originally showing 2–3% associativity misses. Because hits and misses influence the thread order, a lower associativity can sometimes give a lower miss rate, as is the case for '*spmv*' and '*cutcp*'.
- Cache-size is the most important parameter for '*lbm*' and '*spmv*', showing significant miss rate reductions.
- Cache-line size can have both a positive and a negative influence on cache misses. For our benchmarks, a cache-line size of 128B or 256B gives the best results.
- Using only 16 or 32 MSHRs yields better cache behaviour for '*lbm*' and '*spmv*': a low number of MSHRs allows inter-thread locality to be better exploited (see section 4.4). The other benchmarks are not significantly influenced by the MSHR parameter.

8. Summary and future work

This work has shown that reuse distance theory can be used to model GPU caches in detail by extending it with: 1) scheduling of the GPU's threads, warps, threadblocks, cores and sets of active threads, 2) in-flight memory requests and conditional and non-uniform latencies, 3) cache associativity, 4) miss-status holding-registers (MSHRs), 5) and warp divergence. Additionally, micro-benchmarks showed how a Fermi GPU maps addresses to sets in hash-associative caches, and how many MSHRs are available per core.

The new cache model has been evaluated against the Parboil and PolyBench/GPU benchmark suites, comparing modelled miss rates for the GPU's L1 data caches against measured miss rates using hardware counters. The results distinguish different types of cache misses. An example are latency misses, a type not even measured by hardware counters. On average, our model predicts cache miss rates with an absolute error of 6.4% (16KB 4-way) and 8.3% (48KB 6-way). From the 57 tested kernel invocations, 47 lie within a 10% absolute error margin. Compared to the GPU simulator GPGPU-Sim, our cache model shows a better accuracy (6–8% versus 18–21%) and a lower run-time (267x on average). The importance of the discussed extensions become clear when evaluating them separately, showing a reduction in average absolute error when modelling: cache associativity (9.6% $\rightarrow$ 6.4%), latencies (12.1% $\rightarrow$ 6.4%), and a limited amount of MSHRs (7.1% $\rightarrow$ 6.4%).

A more accurate memory latency and warp divergence model can help improve the cache model further, but would require integration with a full GPU execution model. Additionally, the model can be extended to include other GPU caches, such as the L2, the texture caches, or Kepler's new read-only L1 cache. Future work includes the verification of the model on AMD Radeon and ARM Mali GPUs.

References

[1] G. Almási, C. Caşcaval, and D. Padua. Calculating Stack Distances Efficiently. In *MSP: Workshop on Memory System Performance*. ACM, 2002.

[2] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W.-M. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In *PPoPP-15: Symposium on Principles and Practice of Parallel Programming*. ACM, 2010.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *ISPASS: International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009.

[4] K. Beyls and E. D'Hollander. Reuse Distance as a Metric for Cache Behavior. In *IASTED: Conference on Parallel and Distributed Computing and Systems*, 2001.

[5] M. Brehob. *On the Mathematics of Caching*. PhD thesis, Michigan State University, 2003.

[6] X. Chen and T. Aamodt. A First-order Fine-grained Multithreaded Throughput Model. In *HPCA-15: High Performance Computer Architecture*. IEEE, 2009.

[7] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *PACT-19: Parallel Architectures and Compilation Techniques*, 2010.

[8] C. Ding and Y. Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *PLDI-24: Programming Language Design and Implementation*. ACM, 2003.

[9] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44, 2011.

[10] M. Hill and A. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38, 1989.

[11] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *ISCA-37: International Symposium on Computer Architecture*. ACM, 2010.

[12] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests. In *ICS-12: International Conference on Supercomputing*. ACM, 1998.

[13] O. Kayiran, A. Jog, M. Kandemir, and C. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT-22: Parallel Architectures and Compilation Techniques*. IEEE, 2013.

[14] NVIDIA. *CUDA C Programming Guide 5.5*, 2013.

[15] A. Parakh, M. Balakrishnan, and K. Paul. Performance Estimation of GPUs with Cache. In *IPDPSW-26: Parallel and Distributed Processing Workshops*. IEEE, 2012.

[16] T. Rogers, M. O'Connor, and T. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO-45: International Symposium on Microarchitecture*. IEEE, 2012.

[17] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling Performance Variation due to Cache Sharing. In *HPCA-19: High Performance Computer Architecture*. IEEE, 2013.

[18] D. Schuff, B. Parsons, and V. Pai. Multicore-Aware Reuse Distance Analysis. In *IPDPSW-24: Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 2010.

[19] J. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. Hwu. Optimization and Architecture Effects on GPU Computing Workload Performance. In *INPAR: Innovative Parallel Computing*, 2012.

[20] J. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W.-M. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois, 2012.

[21] T. Tang, X. Yang, and Y. Lin. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In *ICDCS-31: Distributed Computing Systems*. IEEE, 2011.

[22] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimisations*, 9(4):Article 54, Jan. 2013.

[23] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *ISPASS: Performance Analysis of Systems and Software*. IEEE, 2010.

[24] W. Xu, H. Zhang, S. Jiao, D. Wang, F. Song, and Z. Liu. Optimizing Sparse Matrix Vector Multiplication Using Cache Blocking Method on Fermi GPU. In *SNPD-13*. IEEE, 2012.