Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors

Sara S. Baghsorkhi Isaac Gelado Matthieu Delahaye Wen-mei W. Hwu

University of Illinois at Urbana-Champaign
Urbana, IL 61801
{bsadeghi, igelado, matthieu, hwu} @illinois.edu

Abstract

With the emergence of highly multithreaded architectures, performance monitoring techniques face new challenges in efficiently locating sources of performance discrepancies in the program source code. For example, the state-of-the-art performance counters in highly multithreaded graphics processing units (GPUs) report only the overall occurrences of microarchitecture events at the end of program execution. Furthermore, even if supported, any fine-grained sampling of performance counters will distort the actual program behavior and will make the sampled values inaccurate. On the other hand, it is difficult to achieve high resolution performance information at low sampling rates in the presence of thousands of concurrently running threads. In this paper, we present a novel software-based approach for monitoring the memory hierarchy performance in highly multithreaded general-purpose graphics processors. The proposed analysis is based on memory traces collected for snapshots of an application execution. A trace-based memory hierarchy model with a Monte Carlo experimental methodology generates statistical bounds of performance measures without being concerned about the exact inter-thread ordering of individual events but rather studying the behavior of the overall system. The statistical approach overcomes the classical problem of disturbed execution timing due to fine-grained instrumentation. The approach scales well as we deploy an efficient parallel trace collection technique to reduce the trace generation overhead and a simple memory hierarchy model to reduce the simulation time. The proposed scheme also keeps track of individual memory operations in the source code and can quantify their efficiency with respect to the memory system. A cross-validation of our results shows close agreement with the values read from the hardware performance counters on an NVIDIA Tesla C2050 GPU. Based on the high resolution profile data produced by our model we optimized memory accesses in the sparse matrix vector multiply kernel and achieved speedups ranging from 2.4 to 14.8 depending on the characteristics of the input matrices.

Categories and Subject Descriptors C.1.2 [*Processor Architecture*]: Multiple Data Stream Architectures; C.4 [*Performance of Systems*] – Modeling Techniques

General Terms Design, Measurement, Performance

Keywords GPU, Memory hierarchy, Performance evaluation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
Copyright © 2012 ACM 978-1-4503-1160-1/12/02. . . $10.00.

1. Introduction

Graphics processors are optimized for throughput oriented workloads, which allowed early GPUs to not include traditional data caches. More recent GPUs have added small per-core L1 caches to capture inter-thread reuse, and larger unified L2 caches to exploit inter-core sharing. This is a significant step forward to better support a more diverse set of workloads and reduce some of the performance discrepancies that previously existed. Yet, support for application developers and compiler designers to investigate the behavior of the source code regions that cause performance bottlenecks has been limited. While current GPUs provide a set of performance counters that collect raw statistics for microarchitecture events such as the overall number of misses for the L1 and the L2 caches, the counter values cannot be monitored or sampled during a program execution. Likewise, support for time-based sampling is provided through application programming interfaces [13] that can be inserted in the source code at the user level. While a time-based sampling built on top of these interfaces can help identify hotspots in a program, a fine-grained sampling approach is not feasible as the high sampling rate will likely distort the accuracy of the profile data. On the other hand, infrequent sampling will increase the number of blind spots and reduce the precision of the measurements.

Profiling the performance of highly multithreaded applications is not confined to recently introduced high performance accelerators. Salapura et al. [16] discuss the significance of the feedback derived from the performance statistics in high performance computing systems such as Blue Gene and propose a new performance counter architecture that scales better with the number of concurrent events in a highly multithreaded system. We propose an alternative software-level approach to capture performance statistics with respect to a highly parallel interleaved memory system and in the presence of a large number of concurrent threads. Our approach does not require the instrumented application to have the same execution timing as the uninstrumented version. The relative ordering of events, e.g., memory loads and stores from concurrently running threads, is reconstructed following a Monte Carlo technique after traces are collected. The Monte Carlo approach allows us to systematically analyze the sensitivity of the predictions with respect to the reconstructed orderings of memory events.

We also propagate the source code locations of memory loads and stores. As a result, we are capable of producing precise and high resolution profile statistics. Most highly multithreaded applications are memory bound and their performance is strongly dependent on efficient use of the memory subsystem. Therefore, in this work we focus on performance modeling the GPU memory hierarchy.

1.1 Related Work

Performance analysis and profiling tools designed for massively parallel systems traditionally targeted coarse-grained instrumentation of multi-threaded environments that use MPI, OpenMP, or a mix of both. Examples of such tools are Vampir [10], TAU [17], and Paraver [14]. Similar to our trace collection mechanism, they usually use a library which when linked and called by an application, collects traces of events. Nevertheless, our trace collection mechanism is completely transparent and requires minimal effort on the developer's part. Furthermore, these tools collect both events information and the time stamps when those events were collected and use the collected time stamps to combine traces from different processes. While this approach introduces little distortion for coarse-grained tracing of events such as entering a shared region, acquiring a lock, or beginning of an MPI data transfer, it cannot be applied to the fine-grained event sampling used in this work for tracking individual memory operations in thousands of concurrently running threads. Our goal is not to construct the exact ordering of the events, rather to set up an execution context similar and close to the actual one; and if we have enough statistical evidence that the reconstructed orderings are close enough to the the actual one, we rely on them to model the inter-thread interactions and interferences.

Previous work in the area of performance tuning and modeling of graphics processors [15, 3, 7, 22] did not consider data caches and the inter-thread interaction of memory operations; they either assumed the memory system is not a bottleneck [15] or modeled a flat simple memory system for the previous GPU generations [3, 7, 22]. These models either combine static analysis and dynamic trace collection via manually inserted probes in the source code [3], or use functional simulators such as GPUOcelot [5] that emulate the GPU execution on a CPU to collect traces. Our parallel trace collection, on the other hand, is transparently integrated into the GPU kernel and incurs much less overhead. The previous work did not provide source-level performance feedback either.

Detailed GPU simulators such as GPUSim [4] and Mult2Sim [21] perform cycle accurate microarchitectural simulations that requires accounting for the timing of microarchitectural events and results in long running simulations. Our approach does not require to keep track of accurate timing of microarchitecture events and therefore incurs less simulation overhead. In addition, our simulation runs (the Monte Carlo random trials) can be executed in parallel. We also provide high resolution source-level performance feedback, which can better guide optimizations.

1.2 Contributions

The main contributions of this paper are:

- We propose a stochastic model of memory hierarchy for highly multithreaded execution environments: While major configuration parameters for the underlying hardware are extracted through microbenchmarking as discussed in Section 2, we use a Monte Carlo approach (randomized trials) to capture the sources of non-determinisim in the simulation model, e.g., the relative arrival ordering of the memory requests at each level of the memory hierarchy. This stochastic approach is presented in Section 3. Experiments presented in Section 5 confirms that the proposed approach yields fast convergence with respect to randomized trials and high accuracy when compared against values read from the hardware performance counters.
- We devise a parallel trace collection mechanism that runs on a GPU and gathers memory traces to capture the intra-thread, inter-thread and inter-core interactions of the memory accesses issued from a set of concurrently running threads. Generated traces are later combined together through the stochastic memory hierarchy modeling framework to predict hit rates for the L1 and L2 caches and measure the expected latency of the main memory. The trace collection is implemented as a source to source compiler transformation module that seamlessly inserts probes inside the GPU kernel (device code) and required routines to handle trace buffers in the surrounding code (host code). The trace collection mechanism, which is discussed in Section 4 incurs modest execution overhead and results in minimal user intervention.
- We keep track of the source code location of memory operations and quantify the efficiency of each individual one with respect to the GPU memory hierarchy. In Section 6, we show how based on the resultant profiling data we estimate the latency of each static load in a program. This information can be used later to target memory optimizations such as tiling and data layout.

2. A Highly Multithreaded Architecture Model

In this section, we define a machine model for a highly parallel graphics processor targeting general purpose and compute oriented workloads. The model must be simple enough to facilitate study and efficient evaluation of the graphics processors. On the other hand, it need to enclose enough details so that it can reflect a realistic behavior of the hardware. In what follows, we highlight a few fundamental differences reflected in this model that separates graphics processors from their peer multi-core processors:

Throughput-oriented Computing: GPUs emphasize high throughput and Single Instruction Multiple Thread (SIMT) performance by having streaming processors running neighboring threads in lock-step and executing the same instruction on different data. This microarchitectural grouping of threads which can affect both control flow and memory access efficiency introduces the concept of *warp* – a group of threads that composes a hardware vector unit.

Hardware Thread Scheduling: Preemptive thread scheduling by the operating system cannot efficiently support a throughput oriented environment. Therefore, GPUs implement thread scheduling through a dedicated hardware. A global thread block scheduler assigns blocks of threads to *streaming multiprocessors*, which are arrays of streaming processors that implement the SIMT execution model. In this work, we will consider a fine-grained round robin scheduling paradigm with respect to the relative order of thread blocks assigned to each streaming multiprocessor. This mapping is the closest model to the dynamic scheduling scheme that the hardware implements. Within a multiprocessor, a warp scheduler alternates between different warps following a weighted round robin scheme.

Bulk Instruction Scheduling: To make the model tractable and avoid fine-grained scheduling of instructions within a GPU kernel, which would result in a full system simulation, we propose a bulk scheduling scheme. The computation within a kernel is divided into a sequence of memory loads followed by a sequence of compute or store instructions. When static memory loads are repeated within a dynamic instruction stream, in the presence of enclosing loop constructs, they end up in separate sequences that are sequentially executed. Therefore, a back-edge in the control flow will terminate a sequence. Another delimitating factor is the existence of data-dependance between static memory operations. For example, if the index

expression of a memory load is computed based on the value that is being read by a previous load, the second load will start a new sequence. Explicit synchronization instructions will also mark the end of the current sequence. Above simplifications preserve enough information about the instruction stream so that the model can reflect a reasonable representation of the kernel computation. Meanwhile, it is general enough that can approximate different instruction scheduling schemes adopted by different GPU vendors. For example, AMD's GPUs are designed based on a Very Long Instruction Word (VLIW) instruction set architecture; multiples of instructions that can execute in parallel are packed into statically scheduled VLIW bundles. These bundles then compose relatively short *clauses*; clauses are initiated by control flow instructions and may only contain a single type of instruction, e.g., memory loads or ALU instructions. In contrast, NVIDIAs GPUs require more sophisticated scheduling logic for fine-grained scoreboarding and resolving dependencies within the dynamic stream of instructions. With the bulk instruction scheduling scheme these level of details are abstracted.

Throughput-oriented Memory System: GPUs conserve memory bandwidth by grouping together, if possible, loads and stores issued within the same warp. Each *coalesced* memory access then uses a single memory request while moving multiple elements of data. Traditionally, highly multithreaded architectures such as HEP [18], M-Machine [6], Tera MTA [19] and more recently multithreaded graphics processors employ hardware multi-threading for fast context switches to tolerate memory latency. Following this approach to tolerate memory latency, the early multithreaded architectures did not include data caches. More recent GPUs such as those based on NVIDIA's Fermi architecture [11] added caches as means to conserve memory bandwidth. Private L1 caches expoit inter-thread data sharing to reduce the memory bandwidth consumption by each core. A shared unified L2 further reduces off-chip bandwidth requirements by exploiting data re-use between cores. The throughput oriented environment also influences the design and functionality of the memory hierarchy. For example, data caches are no longer optimized for exploiting long term locality as their effective size shrinks with the high number of concurrently running threads. These caches are rather designed to exploit short-term spatial and temporal data locaity across the concurrently executing threads.

2.1 Microbenchmarks

We use a set of microbenchmarks to determine memory hierarchy configurations and management policies; in this work we adapt our model to the profile of the Tesla C2050 GPU. Our microbenchmark suite is built around a GPU kernel that traverses elements of one or more linked-lists in a circular (wrap-around) fashion, invoked with a single GPU thread unless stated otherwise.

Cache Parameters: We first measure the cache block size by varying the strides accessing the elements of a linked-list. The experiment initially implies only a cache block size of 128 bytes. This leads one to the conclusion that either there is no higher level caches or the cache block size of the higher level caches are at most 128 bytes. Next, we determine the size of the cache(s) using a fixed stride of 128 bytes (the L1 cache block size) accessing elements of a linked-list. Results from running this microbenchmark on the Tesla C2050 suggest a 16KB first-level cache and a 768KB second-level cache. The Fermi L1 cache can be deactivated though a compiler switch. With that option available, we further investigate the cache block size of the second level cache, which yields an L2 cache block size of 32 bytes. Based on this outcome, we assume that on an L1 miss four consecutive L2 cache blocks are accessed. We also measure that both L1 and L2 caches are 64-way set associative. A 64-way set associative L1 may at first glance look unreasonable, but it is justified with an average measured L1 access latency of 52 cycles; we measured average L1 and L2 access times of 90 and 250 nanoseconds. We later show, in Section 3.2.3, how we customize the measurement of the global memory access time for each application.

Inclusive, Exclusive or Victim: We also need to determine whether the second-level cache is inclusive or exclusive. Therefore, we run a microbenchmark using a linked-list of size equal to the L1 cache capacity for as many thread blocks as the number of available streaming multiprocessors plus one, i.e., 15 thread blocks for the Tesla C2050 GPU. To ensure that different thread blocks are not scheduled at the same time on the same streaming multiprocessor, we enforce each thread block to have the maximum number of in flight warps that can be simultaneously scheduled on each streaming multiprocessor. However, only one thread is performing memory accesses in each thread block. Each of the first 14 thread blocks access elements of 14 completely distinct linked-lists. The last thread block, which is roughly executed after the first 14 finish their execution, randomly chooses one of the previously accessed linked-lists to start its memory accesses. Memory accesses from the first 14 thread blocks will miss in both L1 and L2 caches. However, memory accesses of the last thread block will hit in the L2, if the cache is inclusive and in the L1 if the last thread block is scheduled on the same core as the thread block that has previously accessed the list. We run this microbenchmark 1024 times, and discard the results that indicate the accesses from the last thread block hit in the L1 cache. Next, We run a new experiment: this time all 15 thread blocks access distinct linked-lists. We compare the average memory latency for these two set of experiments. The results indicate a lower memory latency for the case that the last thread block reads from a previously accessed linked-list, which leads us to the conclusion that the second-level cache is inclusive.

Write Policy and Prefetching: Since the L1 caches are relatively small considering the number of inflight threads for each core, it makes sense to forward stores directly to the shared L2 cache. We verified this scenario through microbenchmarking. By closely investigating the numbers reported by the hardware counters, we also verified that stores hit in the L2 only if the corresponding cache block has been activated via a write request, and that the L2 follows a write back, write allocate policy for stores. So for the memory hierarchy model we assume that when a store request reaches the L2, if the cache block is valid it writes to the cache while setting the dirty bit for that block. If the block is not valid, it updates the memory and brings the cache block to the cache and set its valid bit. We also ruled out hardware prefetching by inspecting the number of memory requests reported by the performance counters for microbenchmarks that could potentially trigger hardware prefetching.

3. The Stochastic Memory Hierarchy Model

In this section we describe how a stochastic model of the GPU memory hierarchy is built on top of the deterministic memory model discussed earlier in Section 2. In our stochastic model, the order of memory requests being issued is partially determined by random variables. Memory traces are collected by instrumenting

the GPU kernel at the source code level to record the memory addresses accessed during the kernel execution. The instrumentation framework, which is discussed in more detail in Section 4 is designed as a source-to-source transformation module. Static probes are inserted within the GPU kernel source code to record the memory addresses read from and written to by active threads for a subset of thread blocks as the GPU kernel is executed. This subset of thread blocks represents a snapshot of the kernel execution whose dimensions are determined heuristically based on the number of concurrently executing threads, the average number of dynamic memory operations per warp, and the size of the last level cache.

3.1 Spatial and Temporal Locality – Intra-thread and Inter-thread Interactions

To understand the performance of a GPU kernel with respect to the memory hierarchy, it is necessary to collect enough traces to capture:

- the intra-thread locality and interaction of memory operations
- the inter-thread and inter-thread-block locality and interactions

To capture spatial locality and inter-thread interferences the execution snapshot for trace collection is extended horizontally. We collect traces for thread blocks that are potentially scheduled close together on different cores (streaming multiprocessors), i.e., thread blocks with consecutive logical IDs. To account for the temporal locality and intra-thread interactions the execution snapshot is extended such that enough traces are collected within a single thread. The threshold is determined by the capacity of the last level cache and the total number of inflight threads. If required, the execution snapshot is extended across the boundaries of multiple thread blocks that are scheduled back-to-back on a single core. With this approach, a subset of the memory traces is collected, but the subset is detailed enough to reflect locality and interactions within the group of concurrently running threads.

If required, traces can be collected for multiple snapshots of the kernel execution. However, our initial experiments confirm that applying the model to traces collected from a single snapshot produces precise and accurate enough estimations. This is quite expected as the computation in a typical GPU kernel is structured around a Single Program Multiple Data (SPMD) programming model.

The above approach, is to some extent similar to the time sampling technique introduced by Laha et al. [8]. They used contiguous segments of memory accesses over certain time intervals for trace driven simulation of single-thread workloads. For large caches and in a single-thread execution model, the cold-start error caused due to unknown status of the cache at the start of each trace simulation is a major source of inaccuracy. In a highly multithreaded execution model, the effect of the cold-start error is fairly insignificant; the execution environment setup for throughput oriented caches limits their ability in exploiting long-term temporal locality. We expect that the same limitation will hold in future GPUs as parallelism scales with at least the same rate as the size of the caches increases.

3.2 The Monte Carlo Approach

Collected traces exhibit precise intra-warp (intra-thread) ordering of memory references. But they do not maintain any information on the relative order of memory references issued from different warps or thread blocks. Note that our approach does not rely on the execution order of memory loads and stores when collecting traces. The rationale for not relying on the ordering during the execution of the instrumented GPU kernel is that adding static probes to the kernel source code:

- changes the kernel resource usage (number of registers), which may consequently alter the streaming multiprocessors occupancy, i.e., the number of concurrently active thread blocks.
- increases the number of inflight memory operations, which will distort the state of caches and the level of congestion in the memory hierarchy.
- changes the instruction mix of the GPU kernel and introduces spurious synchronization or stall points.

As a result, instrumenting the kernel will introduce considerable timing distortions in the execution order of the memory references. To account for this effect, the order of memory requests arriving at each level of the memory hierarchy is reconstructed via a Monte Carlo method, which is an efficient sampling approach for systems with individual behaviors highly coupled together. Traces are then driven into each level of the memory hierarchy in our simulator according to the randomly sampled ordering in each run based on the following steps:

1. Given a pool of memory requests waiting to be serviced at each level of the memory hierarchy:
   (a) Generate a valid random ordering from the pool of available memory requests.
   (b) Drive traces to the current memory hierarchy module following the ordering derived in step (a).
   (c) Obtain performance estimations for the current level and prepare the pool of memory request to be serviced by the next memory hierarchy level.
2. Repeat step 1 for a sufficiently large number of times.
3. Determine the probability distribution of results using histograms and summarize the confidence of the predictions.

The output of the model is a probabilistic performance behavior of the memory system such as hit ratios for the first and second level caches. If certain performance behaviors are most frequently observed – even with limited knowledge about the exact relative ordering of inter-thread memory requests – they are statistically sound representatives of the system performance. In other words, if different random orderings result in noticeably different performance statistics (a wide spread histogram) then the predictions are not reliable. Otherwise, though we have not followed the actual inter-thread ordering when driving traces into the simulator, we have set up an execution context similar and close enough to the actual one. In such a case, we evaluate the performance of memory operations within the execution contexts reconstructed following the above steps.

3.2.1 Schedule Deviation

As independent thread blocks are scheduled to run on streaming multiprocessors, their executions start to fall out of sync with each other due to non-uniform memory access latencies, different number of memory operations and computation loads as individual threads or warps may follow different control flow paths, etc. To account for these schedule deviations, when scheduling loads and stores from inflight warps, random start (alignment) points are chosen for thread blocks scheduled simultaneously across different streaming multiprocessors. The horizontal dashed lines in Figure 1 highlight the execution snapshots devised based on random start

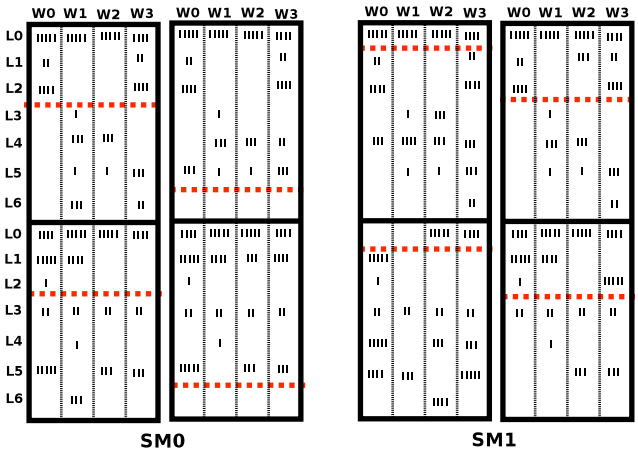

Figure 1. Random points mimic schedule deviation for simultaneously scheduled thread blocks

points for two multiprocessors. In this example, snapshots are expanded across two back-to-back scheduled thread blocks – each composed of four warps: W0, W1, W2 and W3. Each multiprocessor executes two thread blocks simultaneously. Tiny bars in front of dynamic memory operations L0,..., L6 represent individual memory transactions being issued for the corresponding memory vector instructions. The entry is empty if none of the threads within the warp have executed the corresponding load or store.

3.2.2 Scheduling Traces

Following the alignment of simultaneously scheduled thread blocks, we start picking traces from warps within these thread blocks and drive them into the L1 caches. The scheduling scheme used preserves the vector nature of the memory references issued by threads within a warp. Memory references are coalesced based on the size of the data being accessed and the cache block size. Coalesced accesses (memory transactions) for each warp are bound together to ensure that they are all scheduled at the same time. Each warp of a thread block has a queue of ready-to-issue memory transactions. Attached to each memory transaction is a unique ID that corresponds to the source code location of the memory load or store that has triggered the transaction.

Figure 2 illustrates the layout of the stochastic memory hierarchy simulator. The intra-core trace schedulers pick traces from the warp queues following a weighted round robin fashion. Traces are sent to the L1 caches in the order that they have been picked. Loads update the status and counters of the L1 caches. All stores and the loads that miss in the L1 are forwarded to the L2 cache. When scheduling traces from a warp queue, the scheduler continues picking transactions from the same warp if the corresponding cache lines are triggered by static memory operations within the same scheduling sequence, provided that no data-dependence has been recorded between back to back scheduled accesses. This scheduling policy follows the concept of bulk instruction scheduling discussed earlier in Section 2.

The inter-core trace scheduler picks traces from the L1 queues based on a weighted random scheduling algorithm. Loads and stores that miss in the L2 cache along with store evictions are placed into the main memory request queue shown in Figure 2.

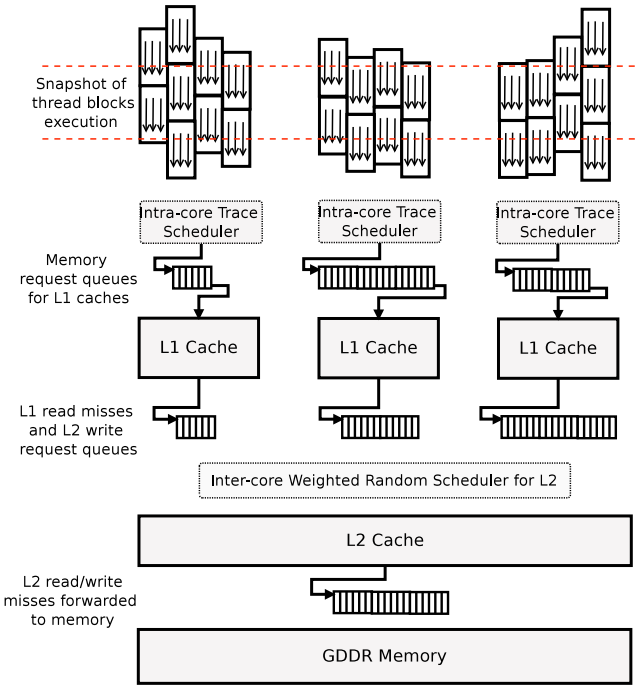

Figure 2. The stochastic memory hierarchy model

3.2.3 Main Memory

In the context of a highly multithreaded environment, memory latency observed from a single-thread execution is not necessarily a meaningful measure of performance. Congestions in the memory system may add to the memory latency. Congestion pattern is characterized based on how efficiently the shared resource is utilized by the memory requests that arrive close in time. The main memory in a GPU is organized into a number of interleaved channels. Depending on how memory accesses are spread across these channels the effective memory bandwidth and latency will change. To estimate the effective memory latency observed by memory operations issued close to each other, memory accesses that reach the main memory are grouped together following a uniform random distribution.

To form groups of simultaneously issued memory loads or stores, random number of memory requests are picked from the queue of misses arrived from the L2 cache. Each group can contain up to 32 memory requests which is the maximum number of distinct memory transactions a core can issue simultaneously, i.e., the memory vector size. Then up to 14 groups are picked, i.e., one per each streaming multiprocessor, to form a set of memory requests that reach the main memory relatively at the same time.

The addresses in these aggregated groups of memory transactions are then normalized and translated into indices of an array whose elements are the size of a single main memory transaction (32 bytes). The indices are then loaded into a microbenchmark and the main memory access latency for each batch of concurrently scheduled accesses is measured via the internal clock register. Figure 3 shows the probability density function of the observed latencies collected by the above approach for the sparse matrix vector multiplication benchmark. The general rule is that more frequent random accesses and higher number of memory bank conflicts will degrade the main memory efficiency and increase the expected latency. If a

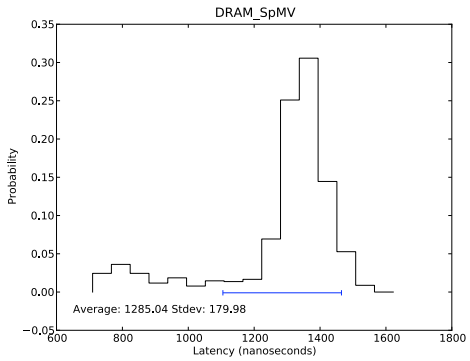

Figure 3. Probability distribution for the main memory latency – SpMV kernel

dominant measured latency exists, we use it as a proper statistical representative latency for the memory accesses that miss in the L2 cache.

4. Software Framework

This section discusses the framework that collects dynamic memory traces for the stochastic memory hierarchy simulator discussed in Section 3.

4.1 Instrumenting the Kernel

To collect and analyze the memory addresses accessed by a given kernel, additional instructions are inserted inside the kernel (the GPU device code) and the CPU host code. This is performed automatically by a source-to-source transformation module that is run before the actual compilation of the program. This module and the actual compiler are wrapped together into a new compiler driver that can replace the original compiler in a project build system. This minimal level of intervention simplifies the use of the framework significantly.

The source-to-source transformation module performs the following source code modifications:

- Inserts *memory address probes* into the kernel source code at locations where the kernel is performing a read or a write operation. A probe as shown in Listing 1 is a call to a function that will be in charge of storing the memory address accessed as well as a unique ID that identifies the source code location of the memory operation.
- Stores for each memory address probe, its exact source code location and the size of the element being accessed into a static array.
- Adds allocation and deallocation routines for the trace collection buffer before and after the launch of the GPU kernel. A pointer to the buffer is added as an extra argument to the kernel invocation code. The organization of the trace buffer is explained later in Section 4.2.
- Adds an additional parameter to the kernel that communicates to the probes the location within the buffer that they need to store the corresponding trace information.
- Adds routines to compute the original kernel occupancy.
- Adds routines that collect traced memory addresses and post-process them for the memory hierarchy simulation model.

```
1  __global__ void SpMV(_sampling_device_buffer *
2      _samples, float *x, const float *val, ...)
3  {
4    _sampling_status  _this_thrd_status;
5    _init_sampling_status(samples, &this_thrd_status);
6
7    tid = threadIdx.y;
8    bid = blockIdx.y;
9    t=0;
10   myi = bid * BLOCKSIZE + tid;
11
12   if ( myi < (numRows) ){
13
14     _sample_mem_index(samples, &this_thrd_status, 0,
15        &(rowInd[myi]));
16     lb = rowInd[myi];
17
18     _sample_mem_index(samples, &this_thrd_status, 2,
19        &(rowInd[myi + 1]));
20     ub = rowInd[myi+1];
21
22     for (j=lb; j<ub; j++) {
23
24       _sample_mem_index(samples, &this_thrd_status, 4,
25          &(indices[j]));
26       ind = indices[j];
27
28       _sample_mem_index(samples, &this_thrd_status, 6,
29          &(y[ind]));
30       yval = y[ind];
31
32       _sample_mem_index(samples, &this_thrd_status, 8,
33          &(val[j]));
34       t += val[j] * yval;
35
36     }
37     _sample_mem_index(samples, &this_thrd_status, 1,
38        &(x[myi]));
39     x[myi] = t;
40
41   }
42
43   _release_buffer(&_this_thrd_status);
44 }
```

Listing 1. Instrumented SpMV – device code

To initiate trace collection for a subset of simultaneously executing thread blocks, one need to know the thread block occupancy for a streaming multiprocessor and the number of available streaming multiprocessors; the latter is obtained via a call to the NVIDIA's programming APIs. The former is computed as follows: The occupancy of a GPU kernel is the ratio of the number of active warps to the maximum number warps supported on a streaming multiprocessor [12]. The occupancy is determined by the amount of resources that the kernel consumes. Resources can be allocated either statically or at the kernel invocation time. Occupancy is also dependent on the GPU device that the kernel will be executed on. Before starting the source-to-source transformation, the driver calls NVIDIA's `nvcc` compiler to collect kernel-specific information required to compute the occupancy, which includes the number of registers and the size of shared memory used. The instrumentation module also inserts calls to NVIDIA's programming interfaces to identify the GPU device that the kernel will be executed on. It then computes the occupancy of the kernel just before its invocation, where it also collects the kernel invocation parameters whose values are required for the occupancy computation, i.e., the thread block dimensions and the amount of shared memory dynamically allocated.

The source to source transformation module is built on top of Clang [1], the C language family frontend for LLVM [9]. Clang has been modified so that it can correctly parse a subset of CUDA [13]

source code and build the abstract syntax tree of both the host and device code. Each node of the tree precisely records its exact location within the source code, even when the node is the result of one or more macro instantiations. This property is essential for the transformation module to provide accurate source level feedback.

The output of the transformation cannot be generated directly from a modified abstract syntax tree: To build the abstract syntax tree, a header containing the declaration of CUDA types and runtime functions must be preincluded. The serialization of the abstract syntax tree would contains these additional declarations. But `nvcc` preincludes a header file as well. So, if `nvcc` is run to compile the source code generated from the abstract syntax tree, it will result in duplicated declaration errors. Therefore, the output of the transformation is a patch file that is applied to the user source code. The modifications that need to be applied to the original source code are recorded as a list of source code insertion operations that are later used to generate the patch file. This mechanism leverages the Rewriter API that Clang provides.

Redundant loads and stores that are likely to be eliminated by the back-end compiler are not instrumented. To achieve this, a global value numbering analysis is implemented to identify the redundant memory accesses and a basic pointer alias analysis is used to determine if a memory operation will be removed through redundant load elimination.

The instrumentation of the source code must be done with an unmodified abstract syntax tree to ensure that the source locations are correct. However, the abstract syntax tree must be modified to increase the quality of the global value numbering analysis: Expression trees are canonicalized to increase the chance that two arithmetically equivalent expressions have the same tree representation. This means that the source code must be instrumented before the analysis is performed. As a result, the source code is over-instrumented. After the analysis is performed, the results are forwarded to the patch file generation module to skip the unnecessary instrumented memory operations.

A significant challenge in instrumenting the memory operations is identifying the correct type of the address space being accessed by a pointer, e.g., global memory, shared memory, etc. The type of a pointer cannot be used to infer which address space it points to as it can change during the lifetime of the pointer. A dataflow-based algorithm is used to disambiguate the address space that a pointer refers to for each segment of the source code. When incapable of resolving the actual pointer type, it conservatively associates the pointer to the global memory, similar to the `nvcc` compiler.

4.2 Recording Traces

The memory address probes inserted within the kernel store the addresses accessed by each thread into a trace buffer. The implementation of these probes and the design of the buffer are inherent to the execution model within the GPU: The threads within a warp are executed in lock-step and the warps within a thread block are executed concurrently. Therefore, the probes are designed to store per-warp rather than per-thread memory accesses. In addition, the buffer is designed so that its size does not limit the acquisition of an unbounded amount of memory addresses.

A trace buffer is composed of a number of block buffers. Each block buffer itself contains a set of warp buffers – one for each warp in the thread block – as illustrated in Figure 4. The number of block buffers is equal to the number of simultaneously active thread blocks in the GPU. We use a per-block-buffer locking mechanism to prevent concurrent accesses to the same buffer from multiple thread blocks. Each block buffer has a counter that holds the number of active threads using the buffer. When zero, the buffer is free to be acquired. A thread block acquires exclusive rights to the buffer when it successfully exchanges, via an atomic operation, the value zero in the counter with the number of threads that it contains. Each thread will atomically decrement this counter just before terminating. A warp buffer is a ring buffer with fixed-sized lines that are wide enough to hold for each thread within a warp, an address value and a static ID associated with the source code location of the corresponding memory operation. When an active thread executes a probe, the next available line in the warp buffer is updated and a line counter is incremented. The buffer is originally initialized with invalid addresses (odd values). So when an inactive thread does not store any address it can be easily captured.

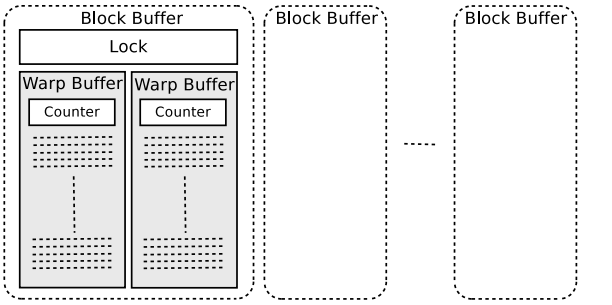

Figure 4. Organization of the trace buffer

Probes will spin on the warp buffer counter when the buffer is full. During the kernel execution, the host program constantly snoops at individual warp-buffer counters. When a warp buffer is full, the host program reads the content of the warp buffer (traces) and resets the warp buffer counter. Now, threads that were spinning on the counter will resume to record traces. When enough traces are gathered, the host program sets the counters to a predefined value to disable probes inside the kernel. The exact layout of the buffer is computed based on thread block dimensions, number of concurrently active thread blocks and the overall size of the buffer itself.

To allow both the host and the device code access the buffer concurrently, the buffer is allocated using pinned-memory. Pinned-memory is the host memory that is removed from the virtual memory, so it is not paged out by the operating system. Since the atomic operations operating on a pinned-memory are not atomic from the point of view of the host [13], the communication between host and device is implemented via un-cached load and store operations by inlining NVIDIA's PTX assembly to treat cached memory lines stale and bypass the GPU L2 cache via write-through stores.

5. Experimental Evaluation

In this section, we present the result of application of the stochastic memory hierarchy model discussed in Section 3 to several GPU applications. The results, which are presented in terms of the L1 hit ratios for loads and the L2 hit ratios for loads and stores are cross validated against the overall hit ratios reported by the hardware performance counters.

5.1 Experiments Setup

The proposed memory hierarchy modeling approach is validated on an NVIDIA Tesla C2050 general-purpose graphics processor. The benchmarks suite chosen covers GPU kernels commonly used in scientific computing and signal processing applications: dense

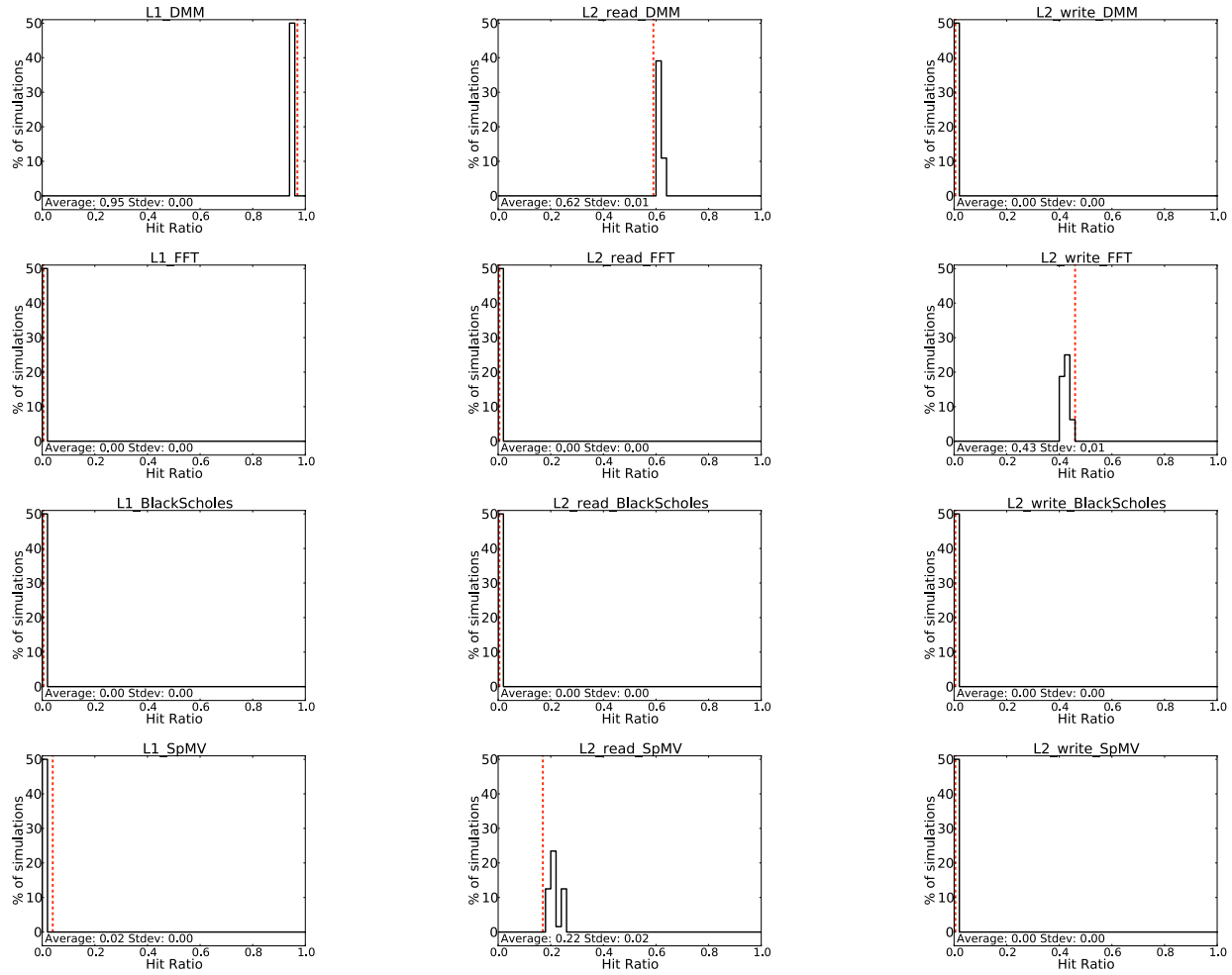

Figure 5. Probability distribution for the L1 read, L2 read and L2 write predicted hit ratios – Part 1

matrix multiplication (DMM), sparse matrix vector multiplication (SpMV), fast Fourier transform (FFT), Black Scholes, the 3D lattice-Boltzmann method (LBM), merge sort (global and shared memory versions), and a 3D stencil computation. These benchmark also exhibit diverse behavior with respect to memory access patterns, data-dependent control flow and memory references, mix of load and store operations, and the locality level available.

The overall probability distributions for the L1 and the L2 cache hit rates are generated by the proposed stochastic model for the discussed GPU benchmarks and results are displayed in Figures 5 and 6. Each histogram shows the probability of a specific predicted hit ratio on the y-axis based on results obtained from 64 simulations. The dashed vertical red line marks the hit ratios reported by the hardware counters.

5.2 Sampling Error – Precision

We report the average and the standard deviation for the predicted hit ratios in Figures 5 and 6. The standard deviation describes the spread of the predicted hit ratios and determines how random variations affects the sensitivity and reliability of the proposed memory model. A small standard deviation – 0.01 on average for the outputs of the model – indicates that the predictions from the model are robust with respect to the randomly sampled ordering of memory requests. Based on the central limit theorem [20] the probability of the expected predicted hit ratios being within the confidence interval of $[\widehat{E}_n - 2\widehat{\sigma}_n, \widehat{E}_n + 2\widehat{\sigma}_n]$ is 95%, where $\widehat{E}_n$ and $\widehat{\sigma}_n$ are the average hit ratios and standard deviations shown in Figures 5 and 6.

5.3 Sampling Overhead

Figure 7 shows changes in the standard deviation of the predictions for the L2 reads in the SpMV kernel as the number of random samples increases; the L2 predictions for loads in the SpMV kernel have the highest standard deviation within our experiments. Nevertheless, a small standard deviation of 0.02 or 2% miss prediction indicates that predictions exhibit a low tendency to be spread out. Based on Figure 7, a relatively fast convergence for the standard deviation (starting after 20 simulations) suggests that larger sample sizes will produce barely noticeable increase in the precision of the predictions specially after 60 simulation runs. For this kernel when run with an input matrix of randomly distributed non-zero elements (51 on average per row), it takes 17.7 seconds to collect traces for an execution snapshot that is 4 times the size of the concurrently running threads on all 14 streaming multiprocessors. It then takes 3.16 minutes on an Intel Core-i7 processor running at 2.8 GHz to finish one simulation and propagate and collect performance statistics. Most of the overhead is in driving traces from the private L1

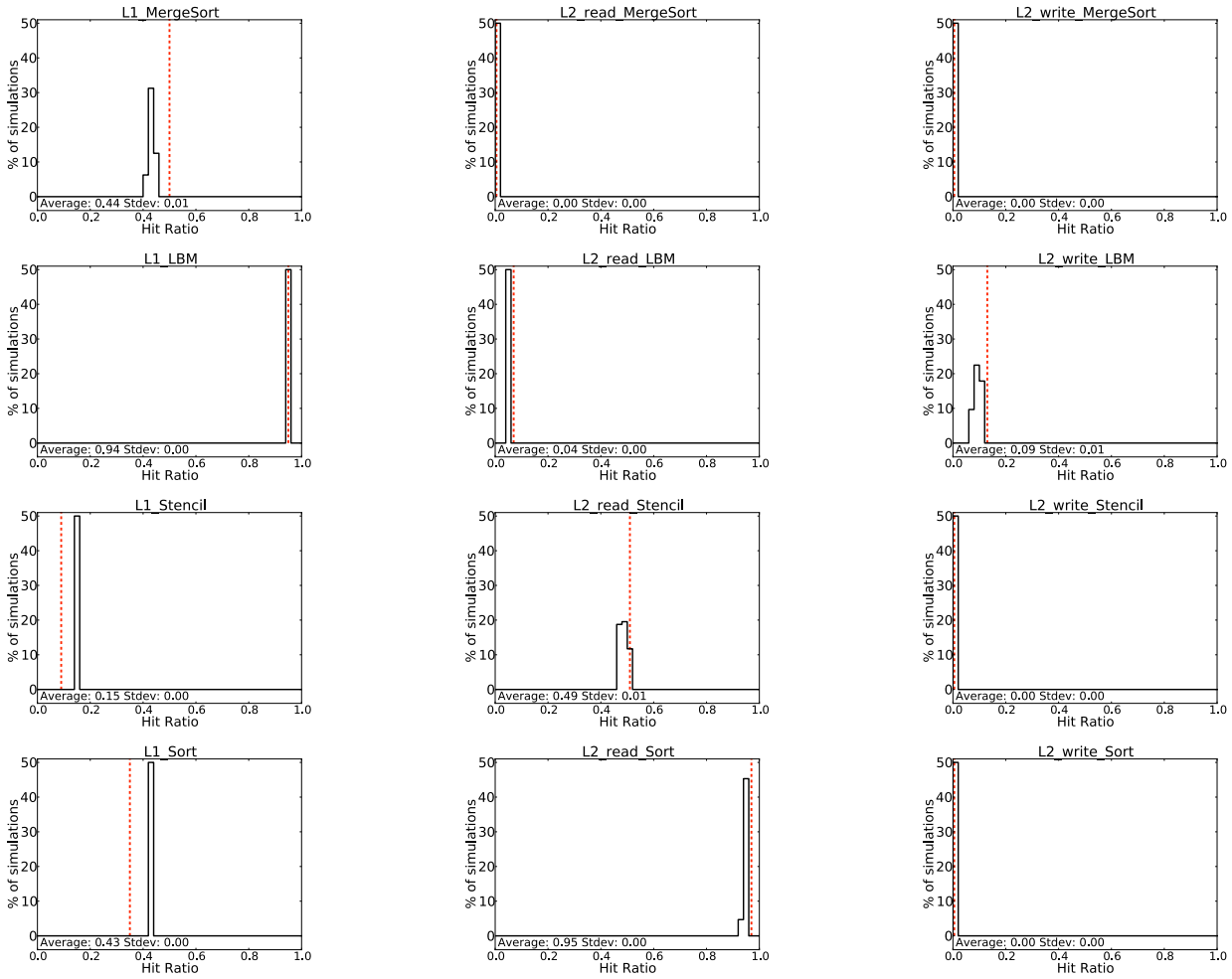

Figure 6. Probability distribution for the L1 read, L2 read and L2 write predicted hit ratios – Part 2

caches to the unified L2 cache. The SpMV benchmark represents the end of the spectrum with slower convergence rates (after 20 simulation runs) for the L1 and L2 hit ratio predictions, at least within the benchmark suite used for our study. In general, a handful of simulation runs converge very quickly or the simulations can stop after the results converge within the desired threshold. In addition, multiple simulation runs can be executed in parallel to reduce the overall simulation overhead.

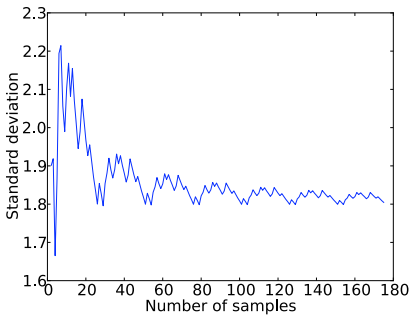

Figure 7. Fast convergence of the standard deviation for the L2 read hit ratio predictions – SpMV kernel

5.4 Systematic Error – Accuracy

We use absolute error to measure the accuracy of our predictions. Relative error is a more meaningful measure of accuracy than the absolute error when the predicted values are large and unbounded. For the predicted hit ratios, relative error does not relate closely to the performance associated with the difference in the predictions. For example, when using the relative error one gets the same value of error for the following pairs of predicted and actual hit ratios: $(0.02, 0.04)$, $(0.2, 0.4)$ and $(0.4, 0.8)$. The fact that the above errors would be judged to have equal importance is not correct from microarchitecture point of view.

We cross-validated the results predicted by the memory hierarchy model with those read from the hardware performance counters provided by the NVIDIA. The performance counters cannot be read or sampled during the kernel execution. Before running the kernel, an environment variable is set to activate the counters. After the kernel execution is finished, the values of the counters that reflect the overall performance statistics with respect to all memory references are written into a log file. The vertical dashed lines in Figures 5 and 6 highlight the hit ratios computed based on values read from the hardware counters. When compared against hardware counters, the proposed approach have an average absolute error of

```
1   float t = 0;
2   unsigned int myi = bid * BLOCKSIZE + tid;
3
4   if ( myi < numRows ){
5       unsigned int lb = rowIndices[myi];
6       unsigned int ub = rowIndices[myi+1];
7       for (j = lb; j < ub; j++) {
8         unsigned int ind = indices[j];
9         yval = y[ind];
10        t += val[j] * yval;
11      }
12      x[myi] = t;
13  }
```

Listing 2. SpMV kernel

3.4% for the L1 read hit ratios, 1.9% for the L2 read hit ratios and 0.8% for the L2 write hit ratios.

6. High Resolution Performance Statistics

In this section, we discuss how high resolution performance statistics are generated by coupling source code level information and the profile data produced by the memory hierarchy model. This comprehensive view of the level of memory efficiency exploited by individual data structures or memory operations within a particular code segment is a crucial step for targeted memory optimizations. We later illustrate though a simple example how this low level information can highlight inefficient memory accesses and play up the data structures that are ideal candidates for optimization.

6.1 The Average Latency per each Static Load

In Section 3.2.3, we presented a microbenchmarking approach to measure the main memory access time, T_m, specific to each application. We also measured the L1 and the L2 cache latencies, denoted by t_1 and t_2, with a simple microbenchmark. We later predicted hit ratios by the proposed memory hierarchy model for the L1 and L2 caches, denoted by H_1 and H_2. In this section, we combine all these results using the following simple access latency equation:

$$X = H_1 \cdot t_1 + (1 - H_1) \cdot [H_2 \cdot t_2 + (1 - H_2) \cdot T_m]$$

In the above equation values of T_m, H_1 and H_2 can change from one simulation run to the other. Furthermore, H_1 and H_2 are the L1 and the L2 hit ratios for a specific static load; since we propagate down the source code location of each memory operation, we can also estimate the hit ratios for individual loads in the program. After substituting the corresponding values from multiple simulation runs, X becomes a random variable. Based on the expected value of X we estimate the expected memory latency of each static load in the kernel source code. For example, Figure 8 shows the the expected memory latency for each of the 5 static loads in the SpMV kernels. This information can be used to target memory optimizations such as tiling and data layout transformation for specific data structures in the program.

6.2 Case Study: Sparse Matrix Vector Multiplication Kernel

In this section we show through an example how the discussed high resolution performance information can be used to apply targeted optimizations. Initial examination of the sparse matrix vector multiplication kernel source code shown in Listing 2 implies that loads from lines 8 and 10 each exhibits a high level of intra-thread temporal locality as they are executed within a tight loop using consecutive indices. Therefore, the latencies for these loads are expected to

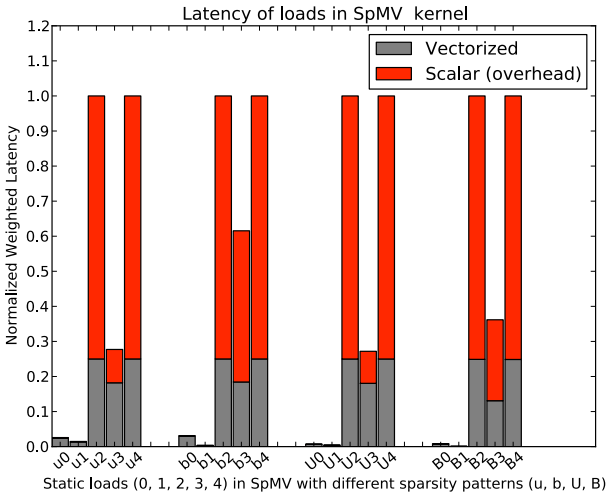

Figure 8. Breakdown of memory latency for static loads in scalar and vectorized SpMV kernels for 4 input matrices: u = an average of 32 non-zero elements per row randomly distributed, b = an average of 32 non-zeros per row with a block diagonal distribution, U = an average of 128 non-zeros randomly distributed, B = an average of 128 non-zeros with a block diagonal distribution.

be low. On the other hand, memory references from the load in line 9 are not analyzable statically and are expected to follow an irregular pattern resulting in poor data locality. Nevertheless, results reported by our memory hierarchy model are different from the speculations derived based on manual examination of the source code. Based on this new information when considering the interactions among the concurrently running threads, loading the vector value *y* in line 9 is more efficient compared to the other two static loads that were expected to exploit a high degree of locality. The very low performance of these loads makes them the most promising candidates for memory optimization in spite of the of the potential overhead that will be introduced.

We applied intra-thread vectorization, a well-known technique to capture temporal locality, to loads from lines 8 and 10. Intra-thread vectorization exploits the data reuse that cannot be efficiently captured by the data caches (the loaded cache lines are evicted before being reused) by packing multiple low performing loads into a single vector load. Listing 3 shows the optimized version of the SpMV kernel, which uses just enough extra registers to maintain the initial occupancy level of the streaming multiprocessors.

We studied the performance improvements of the optimized SpMV kernel with 4 input matrices (u, b, U, and B) differing in the average number of non-zeros per row and the sparsity pattern. Input matrices for u and b configurations have on average 32 none-zero elements while U and B matrices have 128 non-zeros. Non-zeros are randomly distributed in u and U matrices while b and B are block diagonal sparse matrices. Figure 8 shows the predicted load latency for each static load in the scalar and the vectorized SpMV kernels in a stacked format – generated by our stochastic memory hierarchy model – for the 4 resultant configurations. The latencies were normalized with respect to the latency of the least performing load in the scalar kernel for each of the 4 configurations. Based on the profile data shown in Figure 8, reducing the number of inefficient loads in the vectorized configuration will also mitigate the cache trashing effect for vector *y*, specially in case of block diagonal matrices where *y* benefits from a higher degree of locality. The execution times measured for different SpMV configurations

```
1    float t = 0;
2    unsigned int myi = bid * BLOCKSIZE + tid;
3    if ( myi < numRows ){
4     unsigned int lb = rowIndices[myi];
5     unsigned int ub = rowIndices[myi+1];
6
7        //prologue code -- alignment adjustment
8                        .
9                        .
10       for (j = newlb; j < newub; j+=4) {
11          uint4 ind = indices[j/4];
12          float4 value = val[j/4];
13          float yval = y[ind.x];
14            t += value.x * yval;
15          yval = y[ind.y];
16            t += value.y * yval;
17          yval = y[ind.z];
18            t += value.z * yval;
19          yval = y[ind.w];
20            t += value.w * yval;
21       }
22       //epilogue code -- alignment adjustment
23                       .
24                       .
25       x[myi] = t;
26   }
```

Listing 3. Vectorized SpMV kernel

discussed above comply with the profile data shown in Figure 8; vectorized kernels show major performance improvements with speedups of 2.4 (u), 3.63 (b), 2.5 (U), and 14.8 (B).

7. Conclusions and Future Work

This paper presents a novel solution to the problem of providing meaningful performance feedback to developers for highly multithreaded graphics processors. Our stochastic modeling technique allows us to use a simple tracing approach without concerns for distorted execution timing. It further provides error bounds and confidence level information in the presence of scheduling uncertainties and allows us to minimize the cost of tracing and simulation. The close match between the generated predictions and the measured hardware performance counter values provides good validation for our model. The high resolution performance statistics generated through coupling source code level instrumentation and the memory hierarchy simulation provides a comprehensible view of the level of memory efficiency being exploited by individual data structures in the program. This information, though still far from specific optimization hints to a developer or a compiler, is a crucial and fundamental step forward towards that goal. While our model is demonstrated and validated based on CUDA applications and the Tesla C2050 hardware, it is applicable to other highly multithreaded processors. We plan to extend the system to OpenCL [2] applications and the AMD GPUs in the near future.

Acknowledgments

We would like to acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program. In addition, we wish to thank Sam Williams and anonymous reviewers for providing helpful comments.

References

[1] http://clang.llvm.org/.

[2] *The OpenCL Specification*, 2009.

[3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 105–114, 2010.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163 –174, 2009.

[5] G. Diamos, A. Kerr, and M. Kesavan. A dynamic compilation framework for ptx. http://code.google.com/p/gpuocelot.

[6] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 146–156, 1995.

[7] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.

[8] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput.*, 37:1325–1336.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 75–86, 2004.

[10] W. Nagel, A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach. *VAMPIR: Visualization and analysis of MPI resources*. KFA, ZAM, 1996.

[11] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

[12] NVIDIA. CUDA occupancy calculator.

[13] NVIDIA Staff. *NVIDIA CUDA Programming Guide 4.0*, 2011.

[14] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31, 1995.

[15] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

[16] V. Salapura, K. Ganesan, A. Gara, M. Gschwind, J. C. Sexton, and R. E. Walkup. Next-generation performance counters: Towards monitoring over thousand concurrent events. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 139–146, 2008.

[17] S. Shende and A. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287, 2006.

[18] B. J. Smith. Readings in computer architecture. chapter Architecture and applications of the HEP mulitprocessor computer system, pages 342–349. Morgan Kaufmann Publishers Inc., 2000.

[19] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the tera MTA. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–8, 1998.

[20] H. Stark and J. Woods. *Probability, random processes, and estimation theory for engineers*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1986.

[21] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. Oct. 2007.

[22] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture*, 2011.