Statistical Pattern Based Modeling of GPU Memory Access Streams

Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer and Lizy K. John
The University of Texas at Austin, Austin, TX, USA
{reena.panda, xzheng1, jiajunwang}@utexas.edu, {gerstl, ljohn}@ece.utexas.edu

ABSTRACT

Recent research studies have shown that modern GPU performance is often limited by the memory system performance. Optimizing memory hierarchy performance requires GPU designers to draw design insights based on the cache & memory behavior of end-user applications. Unfortunately, it is often difficult to get access to end-user workloads due to the confidential or proprietary nature of the software/data. Furthermore, the efficiency of early design space exploration of cache & memory systems is often limited due to either the slow speed of detailed simulation techniques or limited scope of state-of-the-art cache analytical models.

To enable efficient GPU memory system exploration, we present a novel methodology and framework that statistically models the GPU memory access stream locality. The proposed G-MAP (**G**PU **M**emory **A**ccess **P**roxy) framework models the regularity in code-localized memory access patterns of GPGPU applications and the parallelism in GPU's execution model to create miniaturized memory proxies. We evaluate G-MAP using 18 GPGPU benchmarks and show that G-MAP proxies can replicate cache/memory performance of original applications with over 90% accuracy across over 5000 different L1/L2 cache, prefetcher and memory configurations.

1 Introduction

In the past decade, graphics processing units (GPUs) have emerged as a popular computation platform for applications beyond graphics. Programmers exploit these massively parallel architectures in diverse domains (e.g., linear algebra, bioinformatics etc.). GPUs leverage large amounts of parallel hardware combined with light-weight context switching among thousands of threads to hide the impact of long memory latencies and improve performance. However, many recent studies [8, 13] have shown that the long off-chip memory latencies still limit GPU performance. Hence, on-chip caches have been adopted in mainstream GPUs [1] to reduce the latency impact and off-chip memory traffic. However, GPU cache performance is often sub-optimal due to limited per-thread cache capacity, MSHRs etc. Thus, optimizing performance of GPGPU applications requires evaluating new memory hierarchy designs.

Early design space exploration of GPUs is traditionally done by computer architects and researchers using detailed cycle-accurate simulators [4, 19]. Although accurate, simulators are often very slow, which severely limits the efficiency of extensive design-space exploration [22]. Recently, few researchers have proposed analytical models [15, 21] to estimate GPU cache performance. Although such models are fast, their scope is often limited (model limited degree of parallelism [21], applicable for L1 caches [15, 21]). Furthermore, effective modeling techniques require access to either the application source code or memory traces. Unfortunately, source code or exact memory traces of end-user workloads are often inaccessible due to their proprietary nature [10]. A few examples of proprietary workloads include programs used by the department of energy, national labs, financial applications etc. Therefore, GPU memory system designers need insights into the end-user workloads, but end users can not divulge any proprietary information.

CPU memory system designers have also faced similar problems, and as a solution, they have used a miniaturized representation of the end-user workloads, called a "proxy" or "clone", which mimics the end-user workload performance [3, 10, 18]. However, no such suitable solutions exist for cloning GPU memory access patterns. Most prior GPU performance cloning studies have focused on modeling instruction- or thread-level parallelism [22, 6].

In this paper, we propose **G-MAP**, a novel methodology and framework to statistically model the inherent memory access locality and parallelism of GPGPU applications to create miniaturized GPU memory access proxies. These proxies closely mimic the performance of the original applications and facilitate evaluation of futuristic GPU memory hierarchies. G-MAP exploits three key observations of the GPU execution model to create the memory proxies. First, although GPUs typically run several thousands of threads, the dynamic execution paths taken by most threads can be summarized using a small set of dominant paths. G-MAP leverages this observation to capture a set of dominant dynamic memory execution profiles, which represent the sequence of memory instructions executed by all threads. Doing so also helps G-MAP to later accelerate memory performance modeling by skipping computation instruction processing. Second, individual threads in most GPGPU kernels typically access memory based on a linear (regular) transformation of the thread index. G-MAP exploits this synergy and regularity in GPU memory access patterns to generate proxy memory accesses for each thread, ordered based on the thread's dynamic memory execution profile. Finally, accurate modeling of the cache & memory performance of GPGPU applications requires accounting for GPU's parallel execution model. G-MAP models appropriate thread-level parallelism by leveraging a fine-grained, coordinated scheduling policy to create ordered per-core memory traces from the ordered per-thread access sequences, which can be simulated on detailed multi-core, multi-level cache/memory performance simulators to perform extensive design trade-off analysis.

The combination of statistical profiles captured by G-MAP can accurately mimic GPGPU application memory behavior. Apart from enabling to hide the original memory accesses, G-MAP can scale down the original benchmarks by generating fewer number of accesses in the proxies leading to reduced simulation time and storage requirements. G-MAP may also scale up the original benchmarks to model futuristic workloads with larger footprints, larger number of threads, cores, etc. Other benefits of G-MAP include proxy portability (same statistical profiles collected for CUDA or OpenCL programs). The key contributions made in this paper are:

- We propose **G-MAP**, a novel methodology and framework to statistically model the memory access behavior of GPGPU ap-

DAC '17, June 18-22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062320

plications to create miniature memory access proxies.

- We identify a set of key statistics needed to capture the memory access patterns of GPGPU applications.
- We propose a memory proxy generation and performance modeling technique accounting for GPU's parallel execution model.
- We evaluate G-MAP using 18 benchmarks from Rodinia [5], CUDA SDK [2] and Ispass09 [4] benchmark suites and show that the G-MAP's performance cloning methodology mimics the performance of the original workloads with over 90% accuracy across over 5000 L1/L2-cache/prefetcher/memory configurations.

Without loss of generality, we will refer to GPGPU as GPU in the rest of this paper.

2 Background

This section provides a brief background about the baseline GPU architecture and the GPU application execution model.

2.1 GPU baseline architecture

GPUs consist of a collection of data-parallel SIMD cores (streaming multiprocessors (SMs) in NVIDIA GPUs or compute units (CUs) in AMD GPUs) as shown in Figure 1a. Each SM fetches, decodes a group of threads (warps in NVIDIA GPUs or wavefronts in AMD GPUs) and executes them in lockstep, following the single instruction multiple thread (SIMT) model. GPUs support multiple types of on-chip caches to improve memory bandwidth utilization. Each SM is associated with a private L1 data cache, texture cache, constant cache and shared memory. The global memory is partitioned and all SMs are connected to the memory modules by an interconnection network. Each memory controller consists of a slice of the shared L2 cache and the DRAM partition.

2.2 CUDA/OPENCL execution model

The GPU software execution model is shown in Figure 1b. A GPU application is composed of several kernels. Each kernel is comprised of a grid of scalar threads and each thread has a unique identifier which is used to divide up work among the threads. Within a grid, threads are split into groups of threads called threadblocks (TB) or concurrent thread arrays (CTA). Threads are distributed to SMs at the granularity of entire threadblocks and multiple threadblocks can be assigned to a single SM (if resources permit). Threads in a threadblock are further sub-grouped into warps (where, a warp is the smallest execution unit sharing the same program counter). In our baseline system, each warp contains 32 threads. For memory instructions, a memory request can be generated by each thread and up to 32 requests are merged if these requests are for the cache-line(s). Therefore, only one or two memory requests are generated per warp if requests in the warp are highly coalesced.

3 Related Work

Early design space exploration of GPU memory hierarchy designs is traditionally done using detailed, cycle-accurate simulators [4, 19]. Although accurate, simulator speeds are often very slow which limits efficiency of extensive design space exploration. Few researchers have also proposed analytical models to estimate GPU

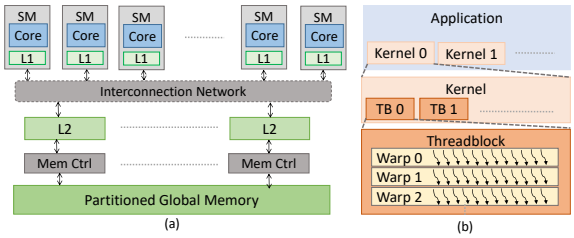


Figure 1: (a) GPU architecture (b) GPU application model

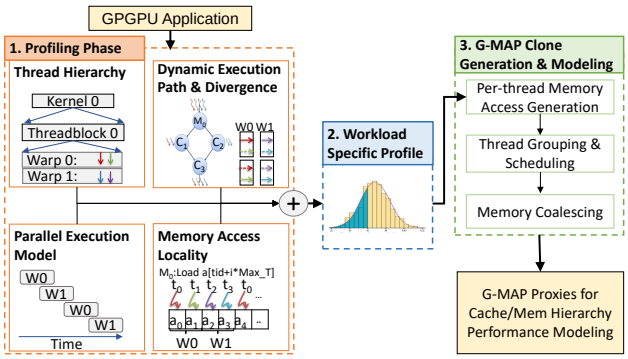


Figure 2: G-MAP framework

cache performance. To model L1 cache miss rate, Tang et al. [21] applied reuse distance theory on a single TB on a single core by arguing that there is limited reuse across different TBs. Nugteren et al. [15] proposed another GPU L1 cache model. They collected per-warp memory traces and emulated inter-warp parallelism using round-robin scheduling policy before applying an extended reuse distance model (considering cache latencies, MSHRs etc.). Although such models are fast, their scope is limited to L1 cache performance modeling. In contrast, G-MAP's performance cloning framework can allow extensive exploration of different levels of the GPU memory hierarchy. Other GPU analytical modeling proposals [8, 20] focus on core performance, while using simple abstractions to model memory performance.

Yu et al. [22] proposed a GPU application cloning technique by replicating the instruction mix, control-flow, divergence behavior etc. Deniz et al. [6] proposed another GPU benchmark synthesis framework by replicating the instruction throughput, compute resource utilization etc. of GPGPU applications. Both the studies focus primarily to mimic instruction-level characteristics and model memory access patterns using abstract/simple models. In contrast, G-MAP models the memory behavior in detail, while abstracting out the core performance. Application cloning techniques have been extensively studied in CPU applications [3, 10, 18, 7], but they are not directly applicable to GPUs [22].

4 G-MAP's Methdology

Figure 2 shows an overview of G-MAP's proxy generation framework. During the profiling phase ①, G-MAP characterizes the GPU application's inherent locality and parallelism patterns (e.g., thread hierarchy, spatial & temporal locality etc.) to create a workload-specific statistical profile ②. We will discuss details of the different profiles captured by G-MAP later in this section. During the clone generation and modeling phase ③, G-MAP adopts a systematic methodology to create a locality- and parallelism-aware clone of the application based on the workload-specific profile, which can be used to drive GPU cache & memory performance exploration.

G-MAP exploits three key features of GPU execution to model memory access behavior using a set of statistical profiles. First, although GPU's execution model supports running thousands of threads, we observed that the ***dynamic memory execution paths*** executed by most threads can be summarized using a small set of dominant profiles. Second, most GPGPU memory operations access memory locations by exploiting a linear transformation based on the *index* (*tid*) of the thread accessing memory. This leads to high degree of regularity in how consecutive threads access different memory locations (***inter-thread locality***) for the same instruction and how individual threads access memory locations during successive iterations of the same instructions (***intra-thread locality***). G-MAP exploits this predictability in both inter- and intra-thread locality to create a memory access trace per thread, ordered based on the thread's dynamic memory execution profile. Third,

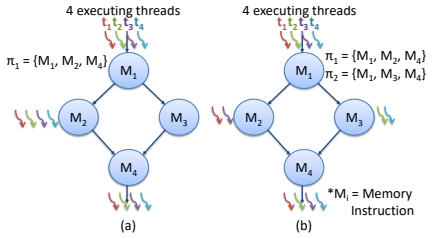


Figure 3: Dynamic memory execution profile capture (a) without and (b) with control-flow divergence

synthesizing ordered per-thread memory traces alone (without accounting for GPU's parallel execution model) is not sufficient to replicate the cache/memory performance. To account for the parallelism, G-MAP leverages ***per-core warp queues*** and a ***coordinated scheduling policy*** to generate ordered per-core memory access sequences from the set of ordered per-thread accesses.

G-MAP maintains the same grid and TB dimensions as the original application. It follows Fermi's [1] execution model to group threads into threadblocks & warps based on section G.1 of CUDA programming guide [16]. G-MAP also implements a memory coalescing model to combine memory requests based on section G.4.2 of CUDA programming guide [16]. Coalescing is modeled before applying the memory locality analysis, as it significantly reduces the computational and memory complexity of the G-MAP model. In the following sections, we will first discuss the profiles collected by G-MAP, followed by the performance cloning algorithm.

4.1 Dynamic memory execution profile

A GPU kernel typically executes thousands of threads. Owing to the CUDA/OpenCL execution model, every thread within a kernel executes the same sequence of instructions (computation & memory) in the absence of control path divergence. G-MAP leverages this observation to capture a single dynamic memory instruction profile (denoted as the π profile) for a *base* thread, as a representation of the sequence of dynamic memory instructions executed by all threads. For example, in figure 3a, all 4 threads follow the same path leading to a single dominant π profile. Of course, this assumption is valid only in the absence of control-flow related divergence effects, which can cause individual threads to execute different paths. We will discuss how G-MAP accounts for such effects in Section 4.4. Nevertheless, the CUDA programming guide recommends writing programs with minimal control-flow divergence as divergence negatively impacts warp occupancies and performance. The π profile is used for synthesizing an *"ordered"* per-thread proxy memory address sequence. G-MAP also exploits code-localization (for every static instruction in the π profile) to capture memory access patterns, as we will discuss next.

4.2 Inter-thread memory access locality

As work distribution in a kernel is primarily done using the *tid* in most GPU applications, GPU memory operations are often a linear function of the *tid* of the thread accessing memory. Since adjacent threads differ by an index of 1, offset between addresses accessed by adjacent threads is often fixed. For example, Figure 4 shows such a kernel with two warps adding two arrays (a,b) under the SIMT model. Here, each warp is composed of 8 threads. We can observe how the consecutive threads access different elements of the two arrays in a regular manner with an inter-thread stride of 1.

Table 1 shows the dominant memory instructions, their frequency, the most dominant PC-localized inter-warp stride (after coalescing requests from threads within each warp) and stride frequency (columns 2-5) across 10 GPGPU applications (benchmark details are provided later). We can observe that across most applications, there exists significant inter-thread memory access regularity for the dominant instructions. G-MAP captures this synergy in memory access patterns across threads in the form of a per-static instruction, inter-thread stride distribution. Later, during proxy generation, G-MAP exploits this information to generate the base addresses of every static instruction executed by each thread, starting from an initial estimate of the base addresses accessed by the *base* thread. Choice of the initial base addresses can help to create obfuscated proxy memory access sequences for proprietariness.

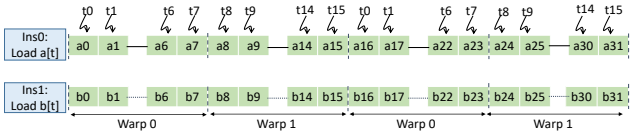


Figure 4: Example showing intra-thread and inter-thread strides with two warps adding elements of two arrays

4.3 Intra-thread memory access locality

Most GPU applications also exhibit regularity in how individual threads access different memory locations during successive iterations of the same instructions (e.g., in a loop). Considering the same example in Figure 4, using its unique *tid*, each thread accesses some elements of the two arrays (e.g., t_0 accesses the 0^{th}, 16^{th} etc. array elements, and other threads follow a similar trend). In all, a thread with tid m accesses $m+(j*\text{Total\_Threads})$ elements of an array (where j represents the currently processed section of data) with an intra-thread stride of 16.

G-MAP exploits this regularity in intra-thread memory access patterns to clone the dynamic memory trace of each thread (memory access ordering is based on the π profile). G-MAP specifically leverages two key intra-thread locality metrics: (a) PC-localized stride distribution and (b) reuse distance. G-MAP captures the distribution of dominant intra-thread strides per PC. Reuse distance is an effective model of temporal locality [14, 15, 21]. It is defined as the number of distinct data elements accessed between the current and the previous access to the same data element. G-MAP tracks intra-thread reuse in the form of LRU stack distance distribution [14] (see Figure 5 for a reuse distance computation example). Table 1 shows the most dominant PC-localized intra-thread stride (after coalescing) and reuse frequency (low, medium, and high reuse implies <30%, 30 - 70% and >70% reuse respectively) across a set of GPU applications (columns 6-7). To synthesize the per-thread proxy sequence, G-MAP generates a memory address for each dynamic memory instruction by first trying to satisfy any dominant intra-thread reuse distance (sampled from the reuse histogram) using an appropriate intra-thread stride value (if possible), followed by sampling a stride value from the intra-thread stride histogram.

4.4 Control-flow divergence

So far, we have assumed that all threads within a kernel execute the same sequence of memory operations, which is represented as the π profile. Even in the presence of control flow divergence, we observed that for most applications, the dynamic memory execution profiles of individual threads can still be summarized using a small set of dominant profiles and their corresponding frequencies (see Figure 3b for an example kernel with two unique π profiles). To do so, G-MAP clusters the dynamic memory instruction profiles based upon their inherent similarity. For a given pair of memory instruction profiles π_i and π_j, their similarity is defined as the total number of identical entries in sequence. Two profiles belong to the same cluster if their similarity is above a certain threshold, Th (Th is empirically chosen as 0.9 in our experiments).

Memory Access	X[0]	X[1]	X[2]	X[3]	X[1]	X[2]	X[3]	X[0]
Cacheline	0	0	1	1	0	1	1	0
Reuse distance	∞	0	∞	0	1	1	0	1

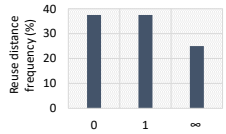


Figure 5: Reuse distance computation example

Table 1: Application memory patterns

Application	Mem PC	%Mem Freq	Inter-warp		Intra-Warp	
			Dom. Stride	%Stride	Dom. Stride	Reuse
Heartwall	0x900	81%	128	51.9%	64	High
	0x4a0	5%	128	51.9%	-128	
	0x4a8	3.8%	128	51.9%	1024	
BP	0x3F8	19.4%	128	75%	128	Med
	0x408	19.4%	128	64.1%	-128	
	0x478	19.4%	128	67.1%	128	
kmeans	0xe8	∼100%	4352	78.2%	-128	High
SRAD	0x250	31.2%	16384	78%	-8192	Low
	0x230	31.2%	16384	75%	-8192	
	0x350	31.2%	16384	80%	-8192	
SP	0xd8	48%	128	88%	4096	Low
	0xe0	48%	128	88%	4096	
CP	0x208	25%	2048	78.2%	-1024	Med
	0x218	25%	2048	78.2%	-1024	
	0x220	25%	2048	78.2%	-1024	
BLK	0xF0	20%	128	77.6%	245760	Low
	0xF8	20%	128	77.6%	245760	
	0x100	20%	128	77.6%	245760	
LUL	0x1c85	4%	352	26%	-128	Low
	0x1ca8	4%	352	26%	-128	
	0x1cc8	4%	352	26%	-128	
LIB	0x1c68	46%	128	57%	19200	High
	0x1ce0	46%	128	57%	19200	
	0x1b40	4%	128	57%	19200	
FWT	0x458	12%	128	88.6%	–	Med
	0x460	12%	128	88.6%	19200	
	0x478	12%	128	88.6%	19200	

4.5 Scheduling policy

Prior research has shown that the order of execution of threads (a.k.a scheduling policy) affects memory hierarchy performance. G-MAP follows Fermi's execution model to determine how threads execute together on a single core. G-MAP assigns threadblocks to cores in a round-robin (RR) fashion until they are full, new TBs get scheduled when the running TBs finish execution. Threads within each TB are sub-grouped into warps and threads within a warp are scheduled simultaneously. To account for GPU's parallel execution model, G-MAP leverages the idea of a per-core warp queue. Initially, the queue is filled with all active warps (from one or more TBs) ordered by the warp identifier (*tid / warp size*). In the simplest form, so long as the queue is not empty, a warp is selected based on RR policy and a single memory request is processed per thread. As a warp finishes a memory request, it is delayed in proportion to the request's latency. This is equivalent to the popular loose round robin (LRR) warp scheduling policy adopted in GPUs. Since G-MAP does not model the detailed GPU core, it captures the effect of other scheduling policies using a simple metric, $SchedP_{self}$, which is defined as the probability of scheduling the same warp consecutively. Although approximate, we will show later that it can estimate cache & memory performance across different scheduling policies. G-MAP models TB-level synchronization by capturing synchronization information in the π profiles and uses it to control the scheduling policy (if needed).

4.6 Proxy generation and modeling

In this section, we will discuss how G-MAP leverages the measured statistical features to generate memory clones for evaluating GPU memory hierarchy performance. Formally, our features can be characterized by a 5-tuple (Π, Q, B, P_S, P_R). $\Pi = \{\pi_1, \pi_2, \ldots, \pi_M\}$ denotes the set of M dominant dynamic memory instruction profiles. Q is a probability measure on Π. $B = \{b(1), b(2), \ldots, b(N)\}$ denotes the base addresses of all N static instructions corresponding to the π profiles.

$$P_S = \{(P_E^{(1)}, P_A^{(1)}), \ldots, (P_E^{(N)}, P_A^{(N)})\}$$

contains a set of distributions $(P_E^{(i)}, P_A^{(i)})$ for each unique static instruction i. Here $P_E^{(i)}$, $P_A^{(i)}$ denotes the distribution of inter-thread stride, intra-thread stride histograms respectively. Finally, $P_R = \{P_R^{(1)}, \ldots, P_R^{(M)}\}$ denotes the collection of reuse distance distribution for each dominant memory instruction profile π.

Algorithm 2 describes G-MAP's proxy generation steps. First,

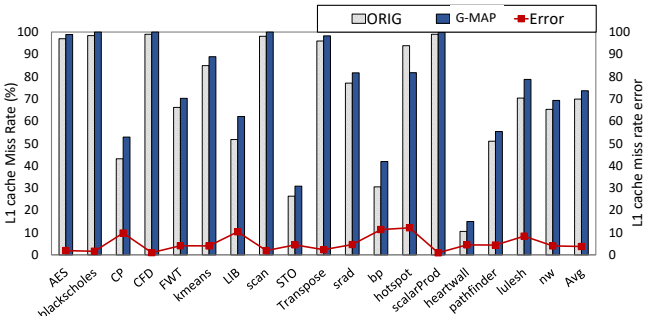
Algorithm 1 Trace Generation for Thread t

1: **Input:** $\pi_i, B, P_S, P_R^{(i)}$;
2: **Output:** $T_t[]$: Memory access for each instruction in π_i
3: **Initialize:** B' = B;
4: **for** j^{th} instruction in π_i **do**
5: $k = \pi_i[j]$;
6: **if** instruction k is being generated for the first time **then**
7: Sample *offset* from inter-thread stride distribution $P_E^{(k)}$;
8: $T_t[j] = b(k)$ + *offset*;
9: $b(k) = b'(k) = T_t[j]$;
10: **else**
11: Sample *reuse* from reuse distance distribution $P_R^{(i)}$;
12: **if** $T_t[j-1-reuse] - T_t[j-1] \in supp(P_A^{(k)})$ **then**
13: $T_t[j] = T_t[j-1-reuse]$;
14: **else**
15: Sample *stride* from intra-thread stride distribution $P_A^{(k)}$;
16: $T_t[j] = b'(k) + stride$;
17: $b'(k) = T_t[j]$;
18: **end if**
19: **end if**
20: **end for**
21: **return** $T_t[]$

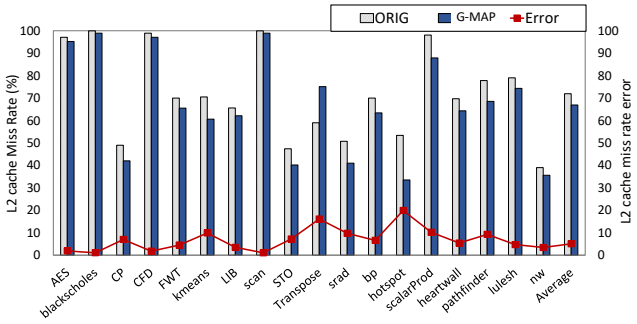
Algorithm 2 Proxy Generation using G-MAP Framework

1: **Input:** Π, Q, B, P, Total number of memory request J;
2: **Output:** $T[][]$: Memory access sequence
3: Determine the number of threads K based on the original application.
4: **for** each thread $t = 1, \ldots, K$ **do**
5: Sample π_i from Π with respect to Q.
6: Generate Trace T_t using π_i, B, P_S and $P_R^{(i)}$. [Algorithm 1]
7: **end for**
8: For each thread t assign its corresponding warp w and core c
9: Perform memory coalescing for all threads in each warp.
10: Let T_w denote the warp-level trace after coalescing for warp w.
11: For each core c, we maintain warp queue WQ_c containing corresponding active warps.
12: **while** $j < J$ **do**
13: **for** $c = 1, \ldots, MAX\_CORE$ **do**
14: Choose a warp w from WQ_c based upon scheduling policy.
15: $T[c][j]$ = T_w.get_next_access(); $j = j+1$
16: **end for**
17: **end while**
18: **return** T[][]

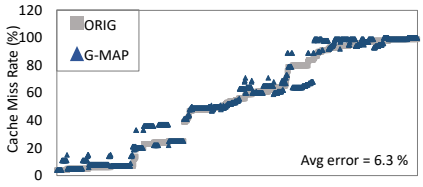
G-MAP assigns a π profile to each executing thread (line 5). Next, G-MAP generates a trace for each executing thread, which is ordered based on the memory execution sequence provided in the π profile (Algorithm 1). To generate the per-thread memory trace, G-MAP uses the inter-thread stride distribution to assign base addresses for the first execution instances of every memory instruction executed by the thread (lines 6-10, Algorithm 1). For successive dynamic executions of the memory instructions, G-MAP assigns memory addresses using the intra-thread stride and reuse locality information as discussed before (lines 11-18, Algorithm 1). Then, G-MAP groups individual threads into TBs and warps based on Fermi's execution model. G-MAP coalesces memory requests of threads within a warp (lines 9-10) to create coalesced warp-level traces. To model the parallel execution model of GPUs, G-MAP exploits per-core warp queues. The queue is initially filled with all active warps ordered by the warp identifier (line 11). To create a unified per-core memory access trace from the ordered per-warp traces (lines 12-17), G-MAP schedules a ready warp from the warp queue and generates an access to the memory hierarchy simulation model for the selected warp's next address (line 15). Finally, the warp queue is updated based on the warp queue maintenance policy discussed in Section 4.5. Miniaturization is performed by scaling down the number of proxy accesses (J), intra-thread statistics followed by the inter-thread statistics by the target scaling factor.



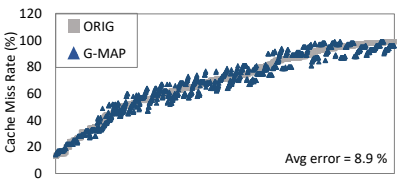
(a) L1 cache miss rate: Varying L1 cache configurations



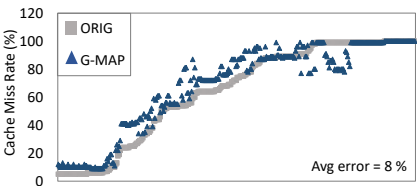
(b) L2 cache miss rate: Varying L2 cache configurations



(c) L1 cache miss rate: L1 prefetcher



(d) L2 cache miss rate: L2 prefetcher



(e) Cache miss rate: Diff. scheduling policies

Figure 6: Evaluating cache, prefetcher and scheduling policy configurations using G-MAP proxies: error in miss rates

5 Experiments and Results

For profiling and validation, we use CUDA-sim (heavily modified for profiling) and GPGPUsim V3.2.2 [4], which is a widely-used cycle-level simulator for GPU architecture research. In order to evaluate a wide variety of real-world GPU applications, we evaluate 18 benchmarks from popular GPGPU benchmark suites like Rodinia [5], NVIDIA SDK [2] and GPGPU-sim ISPASS-2009 [4]. We profile execution of each application until completion or for 1 billion instructions, whichever comes first. It is to note that profiling is a one-time cost and G-MAP receives only a statistical profile as input (independent of the execution length). We choose 1 billion instructions only to keep the evaluation runs manageable. The system configuration used for collecting G-MAP profiles is shown in Table 2. G-MAP proxies are generated with a scaling factor of ∼4-5. For proxy cache and memory performance modeling, we use a validated SIMT-aware multi-core, multi-level cache and memory simulator. The cache simulator is based on CMP$im [9]. Memory system performance is modeled using Ramulator [11], a detailed memory system simulator. We evaluate G-MAP to model performance of L1 data cache, L2 cache and the global memory system. We do not evaluate the performance of shared memory or texture caches, however, G-MAP's methodology is generic enough to capture and replicate patterns in accesses to these caches as well.

We evaluate G-MAP's accuracy in predicting various metrics, including the L1/L2 cache miss rates, prefetcher effectiveness and DRAM performance metrics across ∼290 different configurations per benchmark (over 5000 validation points in all). We use two metrics for validation: the percentage error between original and proxy performance metrics and Pearson's correlation coefficient. Pearson's correlation coefficient indicates how well the proxies track the performance trends of the original applications (1 = perfect correlation, 0 = no correlation). For design space exploration, computer architects care about relative performance ranking, i.e. compare two configurations to see which one performs better. These two metrics together yield how closely the proxies perform with respect to the original workloads across a range of configurations.

Table 2: Profiled system configuration

Component	Configuration
Core Config	15 SMs, 1400MHz, Max. 1024 Threads, 32684 Registers
L1 Cache	16KB 4-way, 128B line size, 1-cycle hit latency
L2 Cache	1MB, 8 banks, 128B line size, 8-way
Features	Memory coalescing enabled, 64 MSHRs/core, LRR sched.
DRAM	GDDR3, 8 Channels, 1 Rank/Channel, 8 Banks/Rank, 924 MHz, tRCD-tCAS-tRP-tRAS: 11-11-11-28, FR-FCFS sched. policy

L1 cache configurations - First, we compare the effectiveness of G-MAP proxies in replicating L1 cache performance of the original applications. We evaluate 30 different L1 configurations per benchmark (varying cache size from 8 - 128KB, associativity from 1 - 16 and line-size from 32 - 128B, while keeping the L2 fixed at 1 MB, 8-way), resulting in over 540 validation points across all benchmarks. The results are shown in Figure 6a. We can observe that the average error between the proxy and original applications is 5.1%. Overall, G-MAP's methodology of capturing both inter- and intra-thread memory access locality leads to high accuracy across most benchmarks. For applications, such as Kmeans and heartwall, which have significant reuse locality, G-MAP's methodology of capturing and replaying reuse distance patterns leads to >97% accuracy in mimicking L1 miss rates. Hotspot experiences the highest error because it does not have significantly dominant intra-/inter-thread stride patterns or reuse locality. Overall, the average correlation between the proxies and original applications is 0.91.

L2 cache configurations - Next, we compare the effectiveness of G-MAP's methodology in matching the L2 cache performance of the original applications (see Figure 6b). Here, we evaluate 30 different L2 cache configurations per benchmark (varying the cache size from 128KB - 4MB, associativity from 1 - 16 and line-size between 64 - 128B, while keeping the L1 configuration fixed at 16KB, 4-way), resulting in over 540 validation points across all benchmarks. Overall, the average error in replicating L2 cache miss rate error is 7.1% and average correlation is 0.91.

L1 cache and prefetcher configurations - Regular access patterns enable prediction of future addresses, making prefetching a viable option [12, 17]. In this section, we evaluate accuracy of the memory proxies in estimating the impact of adding a state-of-the-art L1 prefetcher [12]. We evaluate 72 configurations per benchmark (varying the prefetch degree, prefetcher configurations and L1 cache configurations), resulting in over 1296 validation points. The evaluation results are shown in Figure 6c, sorted according to the original application cache miss rates. Overall, the average error in replicating L1 prefetcher performance is 6.3% and average correlation is 0.9. We observed that scalarProd, srad applications have regular access patterns, still they are largely insensitive to L1 cache prefetching due to larger footprints and lower temporal locality. Hotspot application is also insensitive to prefetching because

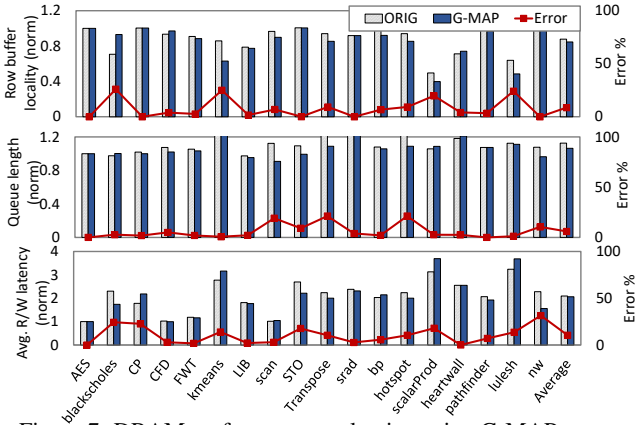


Figure 7: DRAM performance evaluation using G-MAP traces

of non-dominant access patterns and low temporal locality. In contrast, kmeans and nw applications benefit from prefetching.

L2 cache and prefetcher configurations - Next, we compare the effectiveness of the generated proxies in evaluating L2 prefetcher effectiveness. We add a stream prefetcher to the L2 cache and evaluate ∼96 configurations per benchmark (varying the stream window between 8/16/32, prefetch degree between 1/2/4/8 and L2 cache configurations), resulting in 1728 validation points in all. Overall, the average error in replicating L2 cache miss rate error across different cache and prefetcher configurations is 8.9% and average correlation is 0.88 (see Figure 6d).

DRAM performance - Next, we evaluate effectiveness of the memory proxies to enable design-space exploration of the memory system in lieu of the original applications. We use Ramulator [11], a detailed memory system simulator to evaluate 11 different GDDR5 configurations (changing the bus width, channel parallelism, DRAM addressing scheme - RoBaRaCoCh or ChRaBaRoCo) per benchmark (total 198 configurations). We compare three key metrics affecting memory performance: DRAM row buffer locality (RBL), average memory controller queue length and average read/write latency. Figure 7 shows the original versus clone performance values (each value is normalized with original AES's performance metrics) across the 18 benchmarks. Overall the average error in RBL, average queue length and average read-write latency is 9.95%, 8.64% and 12.6% respectively (average correlation = 0.85).

Scheduling policy impact - We also test the effectiveness of G-MAP's methodology in replicating cache and memory performance across two scheduling policies, Greedy-then-oldest (GTO) and LRR (see Figure 6e). As discussed before, G-MAP does not model the GPU cores and it adopts an approximate policy to schedule threads. Nevertheless, the average error in replicating L1 cache miss rate is 8% (5.1% for LRR and 10.9% for GTO policy).

Impact of trace miniaturization - Since G-MAP relies on statistical convergence to replicate memory access patterns, it is important to have sufficient number of samples in the original application to replicate the different probability values due to the law of large numbers. Figure 8 shows the impact of higher degree of trace miniaturization on the performance cloning accuracy (left axis) and speedup of memory simulation using the reduced clone over the full trace (right-axis). We can see that as the trace size is reduced, simulation speed increases almost linearly, while the performance cloning accuracy starts dropping after a certain point. At 8X trace size reduction, the accuracy drops to ∼90%, while the simulation speed improves by ∼8X. The degree of miniaturization on real-world applications can be higher since the number of samples in the real-world application memory traces is often very large.

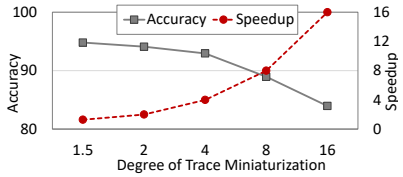


Figure 8: Impact of trace miniaturization

6 Conclusion

In this paper, we proposed G-MAP, a novel methodology that statistically models the memory access behavior of GPU applications by exploiting the synergy in code-localized access patterns (within and across threads). G-MAP also accounts for GPU's parallel execution model by adopting a fine-grained, coordinated scheduling policy to ensure appropriate parallelism at the thread-level and cache/memory-level. We evaluate G-MAP's effectiveness in replicating the memory performance of 18 GPGPU benchmarks and show that G-MAP proxies can replicate the performance of the original applications with over 90% accuracy across more than 5000 different cache, memory and prefetcher configurations, while significantly reducing the simulation time/storage requirements.

7 Acknowledgement

The authors of this work are supported partially by SRC under Task ID 2504 and National Science Foundation (NSF) under grant number 1337393. We wish to acknowledge the computing time we received on the Texas Advanced Computing Center (TACC) system. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other sponsors.

8 References

[1] NVIDIA's next generation CUDA compute architecture, Fermi, 2009.
[2] Nvidia. CUDA c/c++ sdk code samples, 2011.
[3] A. Awad and Y. Solihin. Stm: Cloning the spatial and temporal memory access behavior. *HPCA*, pages 237–247, 2014.
[4] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174. IEEE Computer Society, 2009.
[5] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.
[6] E. Deniz and A. Sen. Minime-gpu: Multicore benchmark synthesizer for gpus. *ACM Trans. Archit. Code Optim.*, 12(4):34:1–34:25, Nov. 2015.
[7] K. Ganesan et al. Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads. *ISPASS*, 2010.
[8] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.
[9] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator.
[10] A. Joshi et al. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *IISWC*, pages 105–115, 2006.
[11] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.
[12] J. Lee et al. Many-thread aware prefetching mechanisms for GPGPU applications. In *MICRO*, pages 213–224. IEEE Computer Society, 2010.
[13] S. Y. Lee and C. J. Wu. Characterizing the latency hiding ability of gpus. In *ISPASS*, pages 145–146, 2014.
[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.
[15] C. Nugteren et al. A detailed gpu cache model based on reuse distance theory. *HPCA*, pages 37–48, 2014.
[16] NVIDIA. Cuda c programming guide 5.5. 2013.
[17] R. Panda et al. Prefetching techniques for near-memory throughput processors. In *ICS*, 2016.
[18] R. Panda, X. Zheng, and L. John. Accurate address streams for llc and beyond (slab): A methodology to enable system exploration. In *IEEE ISPASS*, 2017.
[19] J. Power et al. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE CAL*, 14(1):34–36, Jan 2015.
[20] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *PPoPP*, 2012.
[21] T. Tang et al. Cache miss analysis for gpu programs based on stack distance profile. In *ICDCS*, pages 623–634, 2011.
[22] Z. Yu et al. Gpgpu-minibench: Accelerating gpgpu micro-architecture simulation. *IEEE Transactions on Computers*, 64(11):3153–3166, Nov 2015.