Computing and Informatics, Vol. 38, 2019, 421–453, doi: 10.31577/cai_2019_2_421

View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computing and Informatics (E-Journal - Institute of Informatics,...

RDGC: A REUSE DISTANCE-BASED APPROACH TO GPU CACHE PERFORMANCE ANALYSIS

Mohsen Kiani, Amir Rajabzadeh

Department of Computer Engineering and Information Technology
Engineering Faculty, Razi University
Taghe-Bostan, Kermanshah, Iran
e-mail: {kiani.mohsen, rajabzadeh}@razi.ac.ir

Abstract. In the present paper, we propose RDGC, a reuse distance-based performance analysis approach for GPU cache hierarchy. RDGC models the thread-level parallelism in GPUs to generate appropriate cache reference sequence. Further, reuse distance analysis is extended to model the multi-partition/multi-port parallel caches and employed by RDGC to analyze GPU cache memories. RDGC can be utilized for architectural space exploration and parallel application development through providing hit ratios and transaction counts. The results of the present study demonstrate that the proposed model has an average error of 3.72 % and 4.5 % (for L1 and L2 hit ratios, respectively). The results also indicate that the slowdown of RDGC is equal to 47 000 times compared to hardware execution, while it is 59 times faster than GPGPU-Sim simulator.

Keywords: GPU cache memory, reuse distance analysis, performance modeling, hit ratio

Mathematics Subject Classification 2010: 68M20

1 INTRODUCTION

Many modern high performance computing systems rely on GPUs along with CPUs to deliver high amounts of computing power. Since GPU usage is no longer limited to the graphical processing applications, architectures of modern GPUs are modified towards the benefit of general computations. One of the most significant changes in GPU architectures is the utilization of cache memories in GPUs [1]. Cache memories

can alleviate the traditional problem of memory wall through exploiting the data localities which inherently exist in many general applications. Modern GPUs employ two levels of hardware-managed cache memories. Although cache hit ratios in GPUs are not generally as high as CPUs, the overall GPU performance is highly affected by cache performance in many data parallel applications [2]. In modern CPUs, approximately one-third of the chip area is devoted to cache memories, while the per-core cache size in GPUs is very limited. Moreover, since GPUs use thread switching, a huge number of in-flight threads run concurrently, what results in many attempts to access cache memory lines and causes cache thrashing. Hence, given the limited size of the available cache memory in GPUs, the detailed cache performance modeling is essential. For instance, hardware architects who intend to organize cache memories and application developers who work toward optimum application implementation would highly benefit from detailed cache performance modeling.

There are three main approaches to evaluating how processors function: measurement, simulation, and mathematical performance modeling [3]. To evaluate GPU cache memory performance, appropriate tools and techniques should be developed based on the architectural characteristics of GPUs. It should be noted that the existing CPU cache performance modeling techniques are not applicable for GPUs and require considerable modifications prior to use because of the substantial architectural differences between CPUs and GPUs.

In the present paper, a reuse distance-based approach, called RDGC, is proposed to analyze the performance of GPU cache memory hierarchy. RDGC embodies two models: logical and physical. In the former, the trace information is first extracted, then compressed, and finally ordered logically. In this case, the trace memory accesses ordering is performed regardless of the GPU physical resource limitation, i.e., for unlimited number of processing resources. Hence, the logical model is GPU independent. In the latter case, the physical limitations of a specific GPU, which are essential to the performance estimation of a given GPU, are modeled to define the cache reference sequence. The extended reuse distance (RD) analysis algorithm proposed by the present study is then applied by the physical model to generate the performance metrics for the cache reference sequence. The merit of using these two separate models is that the logical model is not specific to any GPU generation, and its outputs can be used for any GPU machines modeled by the physical model.

Given that GPUs place emphasis on parallelism and the fact that GPU caches may have multiple banks with multiple access ports, RD analysis algorithm [4] was extended in the present study to model such cache memories. In addition, since GPUs employ two levels of cache memories, two cache levels are modeled by RDGC: per-SM (Streaming Multiprocessor) private L1 caches and shared L2 cache.

RDGC provides hardware architects with exploration of GPU cache design space. Additionally, the presented method can be used by application developers to optimize the data locality exploited by cache memories. To analyze the performance of GPU cache memories, different cache design parameters were modeled, including capacity, associativity, block size, bypassing, mapping (indexing) function, and replacement policy. Further, several mapping policies of thread blocks to SMs

were modeled. Finally, the effects of L2 parallelisms were investigated. Moreover, the RDGC was validated against the performance counters provided by NVIDIA's NVPROF profiler. The Polybench/GPU applications [5] and several applications selected from Rodinia benchmarks [6] were executed on a Maxwell and a Kepler GPU, and the results provided by NVPROF were used to validate RDGC. Further, the performance of a selection of applications were evaluated for different cache memory parameters and GPU thread mapping policies.

RD analysis has been adapted for GPU cache memories in studies conducted by Tang et al. [7] and Nugteren et al. [8], in which only a single cache level (L1) is modeled. In this paper, the RD model presented by [8] was extended to include cache parallelism, i.e. multi-port and multi-bank cache memories. In addition, compared to previous studies, the present work is more comprehensive, and two levels of cache memories with different cache parameters are analyzed. Instead of solely generating hit ratios, the transaction counts were also provided by the model as a performance metric which is essential when modeling average memory access time [9]. Furthermore, RDGC was validated for newer GPU generations.

The utilization of cycle-accurate simulators for architectural space exploration is immensely popular with hardware architects. However, simulators are extremely slow and it is exceedingly time consuming to investigate the performance of different cache configurations using a cycle-accurate simulator. The slowdown of GPGPU-sim (V 3.2.2), as a popular simulator, is around 2760000 times for the workload in the present study. Applications with run-times of several milliseconds took hours to be simulated. RDGC has an average simulation slowdown of 47K times, thereby generating the demanded results within several minutes. In addition, the RDGC computations have a degree of parallelism and can be accelerated by parallel programming [10], whereas parallelizing simulators is challenging [11]. In addition, since we use RD analysis, the results of one simulation can be used to predict other cache organizations [12, 9], or it can be employed as a basis to estimate the total processor performance and power [13]. Consequently, RDGC can be used to narrow down the broad architectural space of cache organizations, and the optimal architecture candidates can be later simulated in more details. Last but not least, the older GPU generations are usually simulated by GPU simulators, but their pace of evolution is not in line with GPUs. For instance, the NVIDIA Fermi GPUs are simulated by GPGPU-Sim, while RDGC is designed based on the newer Maxwell GPU generation.

The main contributions of this paper are summarized as follows:

1. A reuse distance analysis algorithm is proposed for modeling the multi-port and multi-bank cache memories. Further, the effects of Miss Status Holding Registers (MSHRs) and cache memory latency are included in the presented model.
2. An appropriate model is developed to model the GPU thread level parallelism and generate the cache reference sequence.
3. Different cache organizational parameters are analyzed in this paper.

4. The effects of L2 cache parallelism in terms of multiple cache partitions and banks are analyzed to quantify the effects of parallelism on the resultant reuse profile.

The present paper embodies the following sections: Section 2 deals with literature review, and a background on NVIDIA GPUs and RD analysis is presented in Section 3. Next, RDGC is explained in Section 4. Later, the evaluation results are presented and discussed in Section 5, and finally, the paper is concluded in Section 6.

2 RELATED WORK

A great deal of studies have been conducted about cache performance modeling in CPUs, whereas the very same subject has not been dealt extensively in the case of GPUs. In the following, we review the related researches to the context of our study.

2.1 GPU Cycle-Accurate Simulators

Hardware architects rely on cycle-accurate simulators to explore the architectural space of GPUs, but its main limitation is the extreme slowdowns of simulators. Although detailed results are provided by simulators, they may fail to provide good insights into some detailed results about the architectural aspects of processors. Further, architectural space exploration with cycle-accurate simulation is very time consuming since it requires to simulate every one of architecture candidates. Some of the prime examples of GPU simulators are GPGPU-sim [14] and Multi2Sim [15].

2.2 Analytical and Empirical Performance Models

An analytical performance model was introduced for GPUs by Hong and Kim [16], but its main problem was that the cache memory effects were not addressed in the proposed model. Later, the said model was extended by Sim and Kim [17] to include the effects of cache memories, which were supposed to be known in advance. In another study performed by Baghsorkhi et al. [18], a hierarchical memory model, based on statistical sampling and trace file analysis, was proposed to predict the performance of the GPU's memory hierarchy. To generate an appropriate memory access sequence, the Monte Carlo simulation method was exploited by the authors of the said study. In another study performed by Huang et al., known as GPUMech [19], the interval analysis technique was extended, in which the parameters affecting the performance of GPUs were modeled, including the effects of multithreading, MSHRs limitation, and DRAM bandwidths. To determine the sequence of memory accesses, the Round Robin (RR) and Greedy-Then-Oldest (GTO) policies were employed. GPURoofline [20] is an empirical approach for performance evaluation and optimization of GPU applications towards observing the performance bottlenecks of applications and manually optimizing the performance of applications. Machine learning was adopted by some researchers to develop predicting performance models

for GPUs. For example, Dao et al. [21] concluded that linear analytical models fail to capture the effects of GPU memory systems and presented a machine learning-based model for GPUs that run the OpenCL kernels to accurately estimate the performance of running kernels. Recently, Kiani and Rajabzadeh proposed a model to approximate the locality in CUDA kernels with regular access patterns [22].

2.3 Reuse Distance-Based Cache Performance Modeling

Multicore CPUs: Although RD analysis is basically designed for single thread analysis, the prevalence of multicore CPUs has motivated many researches toward employing RD analysis for multicore CPUs. Both private and shared caches may exist across a multicore cache hierarchy, each requiring proper mechanisms to calculate the reuse profile. Ding and Chimbili [23] proposed a locality estimation model for multi-threaded applications. The authors modeled thread interleaving and data sharing to profile the locality in shared caches. Similarly, Jiang et al. [24] extended RD profile for shared caches by introducing Concurrent Reuse Distance (CRD) profiles. Their work relies on probabilistic models to estimate CRD profiles from the individual threads memory references. As the authors pointed out, in contrary to RD profiles, CRD profiles are not architecture independent. However, in many applications with similar memory behaviors across threads, CRD can be considered as a virtually hardware independent metric and once acquired for a given architecture, it can be estimated for other architectures [12, 9]. Schuff et al. [25] consider both private and shared caches in multicore systems and extended RD analysis to account for write-invalidation in private and inter-core data sharing in shared caches. Moreover, Wu and Yeung [26] consider loop-level parallelism in which the threads exhibit very similar memory behaviors. The authors used CRD profiles to predict reuse profiles for different core counts in Large-scale Chip Multi-Processors (LCMPs). Their method is useful to conduct core count and problem size scaling analysis. Later Wu and Yeung [12, 9] extended their prior work by employing Private RD (PRD) along with CRD profiles to explore the cache hierarchy architectural space in multicore CPUs. They show that using RD profiles the average memory access time, which is one of the most important performance parameters in CPUs, can be estimated using simple analytical models. Recently, Badamo et al. [13] employed RD analysis and analytical modeling to predict the performance and power consumption and identify power-efficient cache organizations in LCMPs.

Acquiring RD profiles for every possible cache organization is costly thus some techniques have been developed to reduce the analysis time. The first technique is prediction through which the RD profile is acquired for several hardware configurations and then predicted for all other configurations, hence extensively reduces the analysis cost [26]. Another alternative is using statistical sampling methods. In this technique, a small yet representative subset of the

memory references is analyzed which yields a similar profile achievable through full analysis [27]. In addition, RD analysis can be accelerated through parallel execution [10]. It should be noted that applying prediction, sampling, and parallelization techniques is not straightforward in the case of cycle-accurate simulation [11].

GPUs: GPU threads execute the same code (Single Instruction Multiple Threads), thus threads generally exhibit similar memory behaviors. Further, no coherency protocol is employed at L1 level, and only one shared L2 exists in GPUs (see Section 3). Consequently, when adapting RD analysis for GPUs, there is no need to model coherency effects as modeled in multicore CPUs.

Some researchers adapted RD analysis for GPU kernels. In [8], RD analysis algorithm was extended for GPUs to evaluate the performance of L1 cache memory. In addition, the trace file was generated by Ocelot and the access sequence was defined based on the RR scheduling policy. The results demonstrated that hit ratios were chiefly governed by cache capacity, associativity, and block size. However, they do not consider cache level parallelism, and, as the result of the present study shows, cache parallelism can significantly change the achieved reuse distance values. In addition, the authors do not model write accesses and in this article we include writes by considering write-evict policy (which is the policy used in GPUs). Further, Tang et al. also proposed the reuse distance-based algorithm for L1 cache analysis [7]. The problem was divided into two parts. Firstly, a stack (reuse) distance algorithm was developed for a single CUDA block in which the RR policy was assumed for warp scheduling. Secondly, the contention effects, caused by the simultaneous execution of multiple blocks on the very same SM, were modeled. Tang et al., however, do not give any detail regarding the way they modeled the GPU physical limitations. Moreover, they do not model the effects of MSHRs. Recently, RD analysis was employed by Wang et al. to analyze the access patterns of GPU applications [28]. They provide reuse distance breakdown calculated from the memory access information generated by GPGPU-sim. Since their approach relies on GPGPU-sim to generate the memory access information, a considerable time should be devoted for memory trace extraction, thus it is not a time-efficient approach. All in all, RD analysis in the context of GPUs is in its early stages and as a step forward, we try to enhance the existing algorithms by including both cache levels, cache parallelism, and modeling write accesses.

2.4 GPU Cache Memory Organizational Space Exploration

One of the main objectives of the present study is the analysis of the behaviors of cache memories in GPUs in the case of different cache organizations. The organizational space of cache memories and the architectural techniques for cache memories have been investigated in many previous studies [29]. Warp scheduling [30], cache prefetching [31], cache bypassing [32] and cache indexing [33] are among the most

important areas in cache memory organizational investigation that have received a great deal of attention.

3 FUNDAMENTALS: NVIDIA-GPU, CUDA, AND RD ANALYSIS

NVIDIA GPUs: NVIDIA GPUs consists of several streaming multiprocessors (SMs), memory controllers, and an L2 cache memory connected to an off-chip global memory shared between the SMs via an interconnection network. Each SM is composed of processing and memory resources. The former includes processing cores, load/store units, special function units (SFUs), and the latter includes a register file, a shared memory, and an L1 cache. The internal organization of SMs and memory hierarchy varies from one GPU generation to another.

Compute Unified Device Architecture (CUDA): This programming model was developed by NVIDIA for its GPUs towards the development of scalable GPU applications [34]. A CUDA application can be performed on different generations of CUDA-enabled GPUs, possibly with different number of computing resources. In the CUDA programming model, computations are done via several parallel kernels. Each kernel consists of a grid of thread-blocks (blocks for the sake of brevity), where each block is carrying a number of threads. Since no inter-block data dependencies exist in CUDA kernels, the blocks can be executed in any order.

CUDA Memory Model: Logically, in addition to the registers devoted to each thread, each of them possesses a private memory space. A shared memory space is shared between all of the threads in the same block. The global memory is accessible from all of the threads of all blocks.

CUDA Execution Model: When a CUDA kernel is launched on a GPU, the kernel blocks are first mapped onto the GPU SMs. Each SM is capable of performing a given number of blocks concurrently. If the number of mapped blocks exceeds the limit, the extra number of blocks stall until the in-flight blocks are completed. When a block starts executing on a SMs, it is divided into several warps that consist of 32 threads. Ready warps are scheduled onto the available intra-SM resources by warp schedulers. A warp may be stalled, for example, due to a memory reference or an instruction dependency. The number of in-flight warps in a SM is also limited and can further restrict the number of in-flight blocks. The number of warp schedulers and their scheduling policies are different from one GPU generation to another.

3.1 Cache Memory Hierarchy of Maxwell GPUs

In Maxwell GPUs (GM), L1 and Texture caches are integrated. Each SM has a total of 48 kB of L1/Texture cache divided into two 24 kB slices, where each slice is shared by a group of 64 processing cores. In addition to data caching, L1 cache is

also used for register spilling during the execution. L2 cache, consisting of a number of partitions/banks, is shared among all SMs, and all the global memory accesses go to the main memory through the L2 cache. Memory addresses are interleaved among the banks. Maxwell GPUs have several L2 partitions, where each partition consists of two 128 KB banks. An overview of memory hierarchy of Maxwell GPUs (GM) is presented in Figure 1 a). Moreover, the structure and mapping of L2 cache memory of GTX 970, which is used in our evaluations, are shown in Figure 1 b).

L1 caches bypass all the global write memory accesses (a write hit imposes an eviction), while the global read memory accesses can be optionally cached into or bypassed from the L1 depending on the bypassing strategy (not all the NVIDIA GPUs are capable of optional L1 global caching [35]). The bypassing strategy can be defined by compiler flags at the time of compilation.

Miss Status Holding Registers (MSHRs) are a set of registers that track the outstanding missed accesses. A missed access is first compared with the existing content of allocated MSHRs. Consequently, if the requested address is already present in a MSHR (requested by a prior access), the new access will be merged with the existing one, otherwise, a free MSHR will be assigned to the access. When the requested cache block arrives from the backing store, the reserved MSHR becomes free and cache will be filled using the arrived block. The number of MSHRs assigned to each cache is limited, and a reservation fail happens when a missed access does not obtain a MSHR. In this case, the missed access keeps trying to obtain a free MSHR in the next cycles, and the issuing load/store unit stalls during the reservation fail. In addition to the number of MSHRs, the maximum number of per-entry merges is also limited. MSHRs have an important role in delivering the non-blocking cache property, thereby highly affecting the performance of the memory [36]. Accordingly, MSHRs should be modeled as part of modeling the performance of cache memories. Further, it should be noted that the atomic instructions that are handled at L2 level are not considered in the present paper.

3.2 Reuse Distance Analysis

The aim of reuse distance (RD) analysis [4] is to profile the locality of applications. In addition, RD analysis can be used for modeling the cache performance of fully-associative caches with LRU replacement policy. The memory sequences (trace) of accessed cache blocks (or memory addresses) are analyzed to calculate their RDs. The value of RD for a given access to an address is calculated as the number of unique accessed addresses between the current and previous access. Although RD can be calculated with either of memory addresses or cache block granularity, the latter is considered in the present study.

Basically, the main property of RD analysis is its hardware independence. However, for a LRU cache with a given number of blocks, the resulting hit ratio of an application can be calculated based on the RD values of memory accesses. For a fully associative cache with K cache blocks, an access is a hit if its RD value is less than K, otherwise it is a miss. Based on the RD analysis algorithm, the RD value is

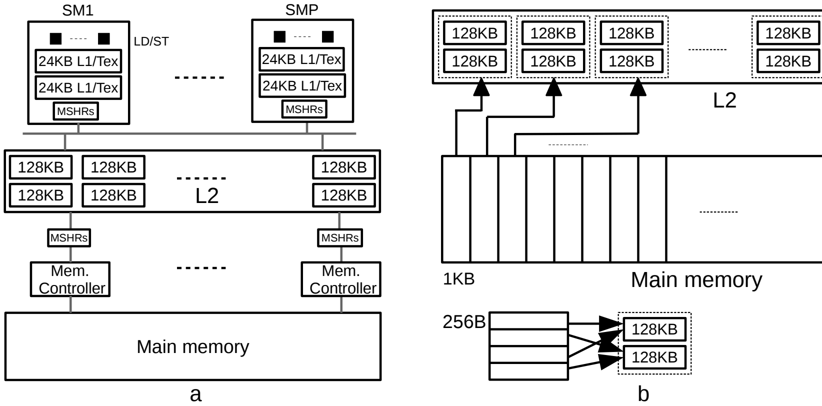

Figure 1. Overview of memory hierarchy of NVIDIA Maxwell GPUs [35], and two-level address mapping scheme of L2 in GTX 970

equal to infinity for the first access to an address, and therefore, access with infinite RD values represent the cold misses. When the intention of RD analysis is hit ratio calculation, a MRU stack is considered that its first block is the most recently used one. An array of counters (denoted by $C[K+1]$), which consists of $K{+}1$ counters, is used to count the hits and misses of the memory accesses, where $C[n]$ holds the number of accesses with RD values of n. $C[K]$ holds the number of accesses with RD $> K$ missing the cache. Given the counter values of a cache with K blocks, the hit ratios of caches with fewer K blocks (e.g., K') can also be calculated through summing up the first K' counters.

As for the set-associative cache memories, the same set of counters can be used for all the cache sets. In the present paper, RD analysis was used to model the performance of set-associative cache memories, and the same set of counters were employed for all cache sets. In Table 1, a typical example is given for RD calculation. In the case of caches with four blocks, the hit ratio equals 50 % without any capacity miss, while for a cache with two blocks, the hit ratio equals 25 %.

Step	0	1	2	3	4	5	6	7
Sequence	A	B	C	D	A	A	D	C
RD	∞	∞	∞	∞	3	0	1	2

The alphabet letters stand for the accessed cache blocks

Table 1. An example for RD analysis

4 RDGC PERFORMANCE MODEL

RDGC, short for reuse distance-based GPU cache, aims to model cache performance. In this method, cache memory hierarchies are analyzed through processing memory access sequences. To do so, the extracted memory trace of parallel blocks are converted into coalesced warp serial access and then analyzed by the RD algorithm that is presented in Section 4.4. In Figure 2, two components of RDGC, namely logical and physical models, are shown.

To analyze the cache performance, the logical model provides the memory access information which is independent of GPU. Then L1 and L2 cache memories are analyzed by the physical model based on the logical trace information along with the physical cache parameters and GPU specifications.

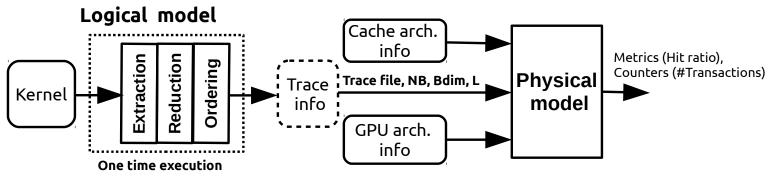

Figure 2. The RDGC components

4.1 Logical Model

To analyze the performance of kernels, the per-thread raw information is first extracted by the logical model. Then the trace file is reduced, and finally the accesses are ordered based on the logical execution model of CUDA. The three phases of the logical model are as follows:

Trace extraction. In the present study, the per-thread memory trace information is extracted through manual probing then executing the kernel. A considerable number of concurrent threads are performed by GPUs. Thus, recording the detailed information for each thread seems impractical and only represents the execution ordering on a specific device. Further, even recording the raw information of all GPU threads at once requires large buffers to store the recorded information temporarily during the trace generation. Consequently, to keep the trace file independent of GPU and to avoid large buffers, the following approach was adopted:

1. Only the raw memory access information (without time stamp) was recorded, and no information was recorded about the thread and instruction ordering and the block to SM mapping. Later, different block mapping and warp scheduling policies can be enforced by the physical model.

2. According to CUDA, blocks can be executed in any order, thus they were separately traced to further alleviate the buffer size. Recording the access information of all the blocks at the same execution run requires a considerable memory space. By separate block trace extraction, the kernel can be launched multiple times, where the information of only several blocks is recorded in each launch.

In Figure 3, the organization of the trace file is depicted, in which each line in the trace file contains the access information of one thread, denoted by ACC. ACC is a five-tuple set in the form of $ACC = \{AN, BID, TID, S, ADD\}$, where

- AN denotes a per-block access number assigned to each access of the block,
- BID is a unique block ID which is assigned to each thread block,
- TID represents the per-block thread identifier,
- S is a Boolean access specifier to define whether the access is a read or a write, and
- ADD denotes the accessed global memory address.

In Figure 3 an example is shown for M blocks and N threads per block ($BID = \{0, \dots, M-1\}$, $TID = \{0, \dots N-1\}$ where L denotes the maximum number of accesses within each thread ($AN = \{0, \dots, L-1\}$). It should be noted that not all the threads within a block necessarily appear in the trace file, e.g., due to a warp divergence.

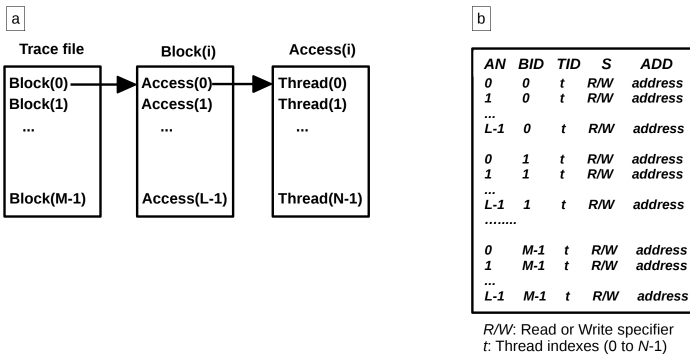

Figure 3. a) The hierarchical structure, b) and overview of the memory trace file

Trace File Reduction. For a kernel with high levels of memory access, the size of the trace file tends to grow rapidly. To alleviate the space overhead of a trace file and to accelerate its processing speed, the generated trace files are reduced through converting the thread access to the coalesced warp access. Moreover, the information of a warp access is stored by each line of the reduced trace file as $\{AN, BID, WID, S, NT, \{ADD\}\}$, where WID is the warp index that is calculated by dividing the thread indexes to the warp size (i.e., 32). Further, NT

denotes the number of active threads of the warp, and the accessed addresses are stored in $\{ADD\}$. As a result, the size of the trace files dropped by 2.3 times in the workload used in this paper.

Trace File Ordering. The memory accesses are ordered by the logical model without enforcing any physical limitations. No GPU related parameters, e.g., warp scheduling policy, are modeled at this step. In the logical model, it is assumed that the accesses to all blocks with the same access numbers (AN) can be executed in parallel with each other. The trace file is ordered according to ANs. Additionally, since the trace files are huge and stored on disks, their ordering is a time consuming operation. In addition to the logically-ordered trace files, grid dimensions (denoted by NB), block dimensions (denoted by $Bdim$), and maximum numbers of per-thread accesses (denoted by L) are generated by the logical model.

4.2 Physical Model

In addition to the trace file information, GPU and cache memory architectural information (listed in Table 2) are received by the physical model. The physical model calculates the MRU counters and then the L1 and L2 cache hit ratios and transaction counts are calculated from the MRU counters. Figure 4 shows the workflow of the physical model. In this figure, $C1R/C1W$ and $C2R/C2W$ denote L1 and L2 cache memory MRU counter arrays for read and write accesses, respectively. Further, the trace files are depicted as dashed rectangles, and the RD analysis, described in Section 4.4, is employed within the physical model to calculate the MRU counters.

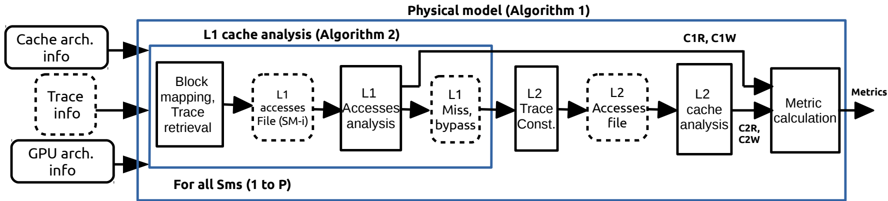

Figure 4. The structure of the physical model

The physical model operates according to Algorithms 1 to 3. As shown in Algorithm 1, the per-SM L1 cache memories are first analyzed (line 3), and the MRU counters (denoted by $C1R$ and $C1W$, for read and write accesses, respectively) are updated accordingly. $C1R$ and $C1W$ are counter arrays with K_1 elements (K_1 is the L1 cache associativity), used cumulatively for all L1 caches. After analyzing the L1 caches, the L2 cache trace file is constructed through retrieving and analyzing the missed or bypassed L1 accesses (line 5) to calculate the L2 MRU counter arrays (denoted by $C2R$ and $C2W$) (line 6). Finally, the output metrics (see Section 5) are calculated based on the MRU counters.

Parameter	Value/Options	Comment
Capacity	S	Capacity of the cache
Associativity	K	Cache's associativity
Block size	B	Block size in bytes
Parallelism	Partition	Number of cache Partitions
	Bank	Number of cache banks per each partition
	Port	Number of ports per each cache bank
Indexing function	MOD	Modulo indexing
	SMODn	Shifted MOD: $[i+m+n,\dots,i+n]$ index bits used instead of $[i+m,\dots,i]$ (m, n are #shifts and index bits [37])
	PRI	Prime modulo Indexing [33]
	XOR	Xor based indexing
Replacement[2]	LRU	Least Recently Used
	LFU	Least Frequently Used
	FIFO	First In First Out
	RANDOM	Random
Bypassing policy	WON	Writes ON, bypass all the write accesses
	RWON	Reads and Writes ON, all accesses are bypassed
	OFF	Bypassing disabled
MSHR	#MSHR sets	Number of MSHR sets
	MSHR Size	Number of MSHRs per each MSHR set
	Max#Merges	Maximum number of merges per MSHR
Resources	P	Number of SMs of the GPU
	n_scheduler	Number of Schedulers per SM
	MAXCW	Max number of in-flight warps
	MAXCB	Max number of in-flight blocks
Blocks-SM mapping	RR	Round-Robin
	BPART1/2	Partitioning, partitions of four/eight blocks
	RAND	Random

[1] Cache parameters are defined for both the L1 and L2
[2] For non-LRU policies, only hit ratio is calculated and the counters do not contain the corresponding RD values.

Table 2. Cache and GPU related inputs to the physical model[1]

4.3 L1 Cache Analysis

The L1 cache analysis algorithm is shown in Algorithm 2. In this algorithm, first, the assigned blocks to the SM are defined based on the given mapping policy (line 1). Then, the access information of the blocks is retrieved from the trace file and stored in a file, called *L1_access* file (line 3), which is analyzed according to the execution model of the GPU to calculate the MRU counters. In the algorithm, B is an array that contains the block indexes of the SM, and BSM denotes the number of blocks

mapped to the SM. As noted before, due to the resource limitations, the number of warps and blocks that can be simultaneously executing on each SM is limited. The maximum in-flight warps and the maximum in-flight blocks per each SM, which are GPU specific parameters, are denoted by $MAXCW$ and $MAXCB$, respectively. Further, CB denotes the number of in-flight blocks on a SM that is defined based on:

1. two kernel-related parameters: grid dimension (denoted by NB) and block dimension (denoted by $Bdim$);
2. two GPU-related parameters: maximum number of in-flight blocks (denoted by $MAXCB$) and maximum number of in-flight warps (denoted by $MAXCW$).

Algorithm 3 is used to define both CB and the number of iterations (denoted by $Nitr$) required to analyze all the blocks. The number of blocks of the SM (denoted by BSM), is defined based on the chosen blocks to SM mapping policy. In Algorithm 2, when CB and $Nitr$ are defined through invoking Algorithm 3 (line 4), the analysis is performed $Nitr$ times, each time for the maximum of CB blocks, through retrieving and processing the information of the in-flight blocks. In each iteration, the order of accesses is the very same order defined by the logical model.

To analyze each access within an inflight-block, the information with memory address granularity is converted to a coalesced cache block access. The coalesced access information of one or more warps (depending on the number of schedulers per SM, $n_scheduler$) is stored within a list (denoted by W), and then the RD analysis is applied to W (line 11). Once all in-flight blocks are analyzed, they will be retired and a new set of blocks (if any) will be processed (line 5 to 15) to analyze all BSM blocks of the SM.

Due to space limitation and its similarities to L1 analysis, the L2 analysis algorithms are not covered here.

Algorithm 1: Physical model

```
Input: Trace, cache parameters, GPU specification
Output: Metrics
  initialize();
  for sm := 1 to P do
    L1_cache_analysis(sm, Trace); /* Update C1R, C1W. Algorithm 2 */
  end for
  L2_trace_construction();
  L2_cache_analysis(); /* Calculate C2R, C2W */
  Metrics = calculate_metrics(C1R, C1W, C2R, C2W);
```

4.4 RD Calculation for Parallel Cache Memories

In this section, an RD analysis algorithm is proposed for parallel caches with multiple banks and access ports. Since GPU hardware parameters affect the resulting RD

Algorithm 2: L1 cache analysis

Input: sm, Trace,
Output: Update $C1R$, $C1W$

1: $\{B, BSM\} = block_mapping(sm, NB)$; /* Define the blocks of the SM */
2: $WpB \leftarrow \lceil \frac{Bdim}{Wsize} \rceil$
3: $L1_accesses = trace_retrieval(B, BSM, \text{Trace})$;
4: $\{CB, Nitr\} = define_concurrent_blocks(WpB, BSM)$; /* Algorithm 3 */
5: **for** $i := 0$ to $Nitr - 1$ **do**
6: $get_inflight_trace(B, L1_accesses)$; /* retrieve access info of the in-flight blocks */
7: **for** $j := 0$ to $L - 1$ **do**
8: **for** $k := 0$ to $CB - 1$ **do**
9: **for** $l := 0$ to $\lceil WpB/n_scheduler \rceil - 1$ **do**
10: $W = create_access_list(i, SB[k], l)$;
11: $RD_profile(W)$; /* Section 4.4 */
12: **end for**
13: **end for**
14: **end for**
15: **end for**

Algorithm 3: Defining concurrent blocks

Input: WpB, BSM
Output: $CB, Nitr$

1: $CB \leftarrow BSM$
2: **if** $BSM > MAXCB$ **then**
3: $CB \leftarrow MAXCB$
4: **end if**
5: **if** $WpB \times CB > MAXCW$ **then**
6: $CB \leftarrow \lfloor \frac{MAXCW}{WpB} \rfloor$
7: **end if**
8: $Nitr \leftarrow \lceil \frac{BSM}{CB} \rceil$
9: **return** $CB, Nitr$

values, it is no longer a hardware independent algorithm. As explained above, the physical model properly generates the warps to cache access sequence. Therefore, the cache reference sequence is known in this stage, however, the sequence is not pure serial and satisfies cache level parallelism (several warp schedulers issue coalesced accesses). The coalesced accesses are mapped onto the cache banks and ports. The proposed RD analysis method is similar to the method introduced in [8]. The following summarizes the differences of the present study with the mentioned work.

- Nugteren et al. modeled serial caches. However, cache level parallelism can change the achieved RD profile (see Section 5.3.3) and the present work included cache level parallelism.

- The authors only modeled L1 cache while both L1 and L2 caches are included in our model. Further, more cache related organizational parameters are investigated in our work.
- Since L1 cache bypasses the write accesses, Nugteren et al. only considered read accesses. However, for write-evict policy, ignoring the write accesses can cause considerable errors in write intensive applications. In the present study, both read and write accesses are included and L1 either caches read accesses (enabled) or bypasses read and write accesses (disabled). When enabled, L1 follows the write-evict policy [34].
- Nugteren et al. assumed that a reservation fail cancels the failed access while other accesses of the same warp, possibly from later instructions, can proceed. This means that load/store instructions may be executed out of order, which is not realistic. In this paper, like some other researchers [36], a reservation fail stalls the warp until all the accesses of the warp are serviced.
- In the mentioned research, the notion of latency miss is introduced to count the event in which a miss encounters a pending previous miss to the same cache block (which exists in a MSHR). In this article, since such requests are merged into the existing MSHR, this parameter is equal to the number of merged requests.
- The probabilistic latency model introduced by the authors can repeatedly change the access order while, as described in the following sub-section, the adaptive latency model can produce smoother and more realistic latency values for the missed accesses.

In Table 3, an example of RD calculation is shown for three warps that access {A, B, C, D}, {E, F, G, H}, and {A, D} cache blocks, respectively. Each cache block is mapped to one of the cache banks. The assumed cache has two banks (B_0, B_1), each having two ports (P_0, P_1), and two MSHRs are shared between the banks. Further, it is supposed that A, C, E, and G are mapped to $B0$ and the other blocks to $B1$. Note that RD is calculated for each bank separately. The first row of the table shows the steps of RD calculation. The next three rows demonstrate the accessed blocks that mapped to each cache bank/port and their corresponding RDs. In addition, the fifth row shows the number of free MSHR entries and the next two rows illustrate the corresponding status of each access (hit (h), miss (m), or reservation fail (rf)). Finally, the last row represents the updated cache blocks which is done by the arrived blocks from the backing store. It is assumed that there are two warp schedulers that coalesce and issue the warp accesses.

Each step of RD calculation includes two phases. In the first phase, the requested blocks are mapped onto the cache banks (according to a given mapping scheme) and access the banks through the available ports (if any). If an access misses the cache, a MSHR entry is reserved when possible, otherwise (denoted by rf in Table 3), the failed access will keep trying to reserve an MSHR in the subsequent steps.

In the second phase, the state of the cache is updated by the cache blocks arriving from the backing store, their assigned MSHR entries become free and update the

cache state so that they are available in the next steps. It should be noted that a hit access also causes some cache updates. In Table 3, the latency of missed access equals two steps. Hence, the requested block by a missed access in the ith step arrives at the end of the step $i+1$ and updates the cache state. As a result, this block will be available from the step $i+2$ forward. It should be noted that 'step' denotes a virtual notion and is not the same as clock cycle. It can be used, nevertheless, as a performance criterion in RD calculation.

The number of cache banks and access ports of each bank can alter the resulted RDs, thus their inclusion in the model is necessary. In the proposed algorithm, to define the exact warp sequence, the following assumptions are considered:

- Warps with smaller indexes have higher priorities in accessing banks and MSHRs.
- Warp schedulers stall until all issued accesses are resolved [36].
- In each step, multiple accesses can be inserted to or removed from the MSHRs.
- Multiple cache blocks can arrive from the backing store within the same step and fill the cache at the end of that step. In this case, the cache state is updated within the same order that the arrived access has been inserted into the MSHRs. This order affects the subsequent RD calculations.

Step		0	1	2	3	4
B_0	P_0	A	E	E	–	A
	P_1	C	G	G	–	–
B_1	P_0	B	B	F	F	D
	P_1	D	D	H	H	–
		∞	–	∞	–	3
RD		∞	–	∞	–	–
			∞	–	∞	2
			∞	–	∞	–
#Free MSHR		0	0	0	0	2
Status, B0		m	rf	m	–	h
		m	rf	m	–	–
Status,B1		rf	m	rf	m	h
		rf	m	rf	m	–
Update		–	A	B	E	F
		–	C	D	G	H

Table 3. An example of RD calculation in Parallel cache memories

4.4.1 The Latency Model for RD Calculation

In the RD analysis algorithm, an appropriate model is required to properly define the latency of missed accesses, based on which the cache state is updated. The values of access latency within a real GPU depend on many parameters, e.g., the instruction

mix, memory access pattern, and the L1-L2 and L2-main memory bandwidths. The probabilistic model used by Nugteren et al. can produce substantially different latency values for two close accesses and even may re-order them, thus we ignore this model. Instead, two types of latency models are tested in the present study. The first latency model sets the latency of the missed accesses to a fixed value, whereas the second is an adaptive model that calculates the latency values based on some dynamic run-time statistics and can provide smoother and more realistic latency values than a probabilistic model. For L1, the latency is calculated by the adaptive model as $K_1 + K_2 \times \frac{\#MSHR_Busy \times \#active_SMs}{L2P}$, where K_1 and K_2 are constant values, $\#MSHR_Busy$ represents the number of outstanding misses, $\#active_SMs$ is the number of active SMs during the execution, and $L2P$ denotes the L2 cache parallelism. The constant values should be defined according to the GPUs data transfer bandwidths. It should be noted that the possible bottlenecks are ignored in the proposed model at L2-main memory transfers. A similar model can be derived for L2. We performed an analysis to investigate the effects of the mentioned latency models on the resultant performance parameters and presented the analysis results in Section 5.1.

4.4.2 L1 and L2 Cache Parallelism Modeling

In the present paper, the default configuration of L1 caches were a double-ported single-bank caches with a set of 32 MSHRs and the maximum number of eight merges per entry. Further, L2 cache was modeled based on the organization shown in Figure 1 (b), and four MSHR sets with 32 entries were used for L2. The first three MSHRs sets were assigned to the first six L2 partitions (one MSHR set shared between two partitions) and the last MSHR set was assigned to the last partition.

5 RDGC EVALUATION

In the present study, mainly, Polybench/GPU benchmark suite [5] is used as the main workload. In addition, several cache intensive kernels were included from Rodinia [6]. Polybench/GPU kernels immensely rely on the hardware managed cache memories thus put more pressure on the cache hierarchy, which is the focus of this paper. On the other hand, most other benchmarks heavily used shared memory and thus most of the data transfers are handled by the shared memory. Consequently, the hardware managed caches are only used to transfer the required data to shared memory. As a result, such benchmarks may fail to properly stress the L1 cache and especially L2 cache, which is an order of magnitude bigger than L1 caches. It should be noted that, none of the used benchmarks utilized atomic instructions, and texture caches. The benchmarks with their main specifications are listed in Table 4. RDGC evaluation was performed within three steps. In the first step, the latency models introduced in Section 4.4.1 were tested (Section 5.1). In the next step, RDGC was validated by comparing its outputs with the values recorded by the performance counters (accessed through NVPROF) on two GPUs including

a Kepler GT 740M and a Maxwell GTX 970 (Section 5.2). In the last step, different cache organizational parameters, including cache capacity, associativity, block size, mapping functions and replacement policies, were evaluated. In addition, multiple blocks to SM mapping policies were evaluated and, finally, the effects of multiple L2 cache parallelism levels on the achieved RD values were analyzed (Section 5.3).

The outputs are provided by RDGC as several metrics. It should be noted that in GT 740M GPU, L1 is disabled for both load and store accesses. Further, in GTX 970 GPU, the texture cache (denoted by tex in the figure) is the same as L1 cache. As mentioned earlier, two compiling options are available for GTX 970: "`-Xptxas -dlcm=cg`" option to disable L1 and "`-Xptxas -dlcm=ca`" option to enable the L1, which in this case L1 only caches the read accesses. Although NVPROF provides a metric to represent the texture cache hit ratios, this counter also counts the other non-workload accesses, e.g., register spilling. Hence, this value is not the exact value of the hit ratios of the requested workload data. In this work, L1 hit ratios are calculated indirectly from other counters. The same phenomenon also occurs at L2 level. Typically, since L2 is significantly greater than L1 and the fact that most of the non-workload traffics are filtered at L1, the resultant errors are likely to be negligible. The brief explanation of the output parameters calculated by RDGC is as follows:

- $L1_R_Hit$ is the read hit ratio of L1, when L1 is enabled.
- $L2_Hit$ is the hit ratio of L2 when L1 is enabled.
- $L2_Hit_L1B$ is the hit ratio of L2 when L1 is disabled (bypassed).
- $L1_R_Trans$ is the read transaction count of L1, when L1 is enabled.
- $L2_Trans$ is the transaction count of L2 when L1 is enabled.
- $L2_Trans_L1B$ is the transaction count of L2, when L1 is disabled.

5.1 Latency Models Evaluation

In this section, the latency models, including the fixed and adaptive models introduced in Section 4.4.1, are tested to reveal the effects of latency on L1 cache performance. The analyzed system has eight SMs, 32 KB of four-way set-associative LRU L1 caches with 32 B blocks, and a set of 32 MSHRs per L1 with maximum of eight per-entry merges. Figure 5 shows the effects of latency on a) hit ratios, b) reservation fails, c) MSHR address merges, and d) steps variations in RD calculation. For the adaptive model, four different values of (K_1, K_2) are tested including $(1, 0.125)$, $(2, 0.25)$, $(4, 0.5)$ and $(8, 1)$ which are denoted by A1 to A4, respectively. Furthermore, to observe the RD calculation performance, the variability of steps in RD calculation is also given in the figure with respect to the number of step counts of the fixed latency with value of one. As can be seen, by changing the latency, the resultant merged and reservation fails are significantly changed, however, hit ratio is not witnessed extreme changes. In the rest of this paper, A2 model is used (the constant values are defined based on the GPU specifications).

Application	Size (N)	Kernels	L	RDGC slowdown (10^3)	GPGPU-Sim slowdown (10^3)
2DCONV†	4 096	2DCONV	10	212	3 084
2MM†	384	K1	$3N$	59.2	1 068
3DCONV†	256	3DCONV	$12N$	69.2	3 409
ATAX†	4 096	K1,K2	$1+3N$	56.4	6 546
BICG†	4 096	K1, K2	$1+3N$	45.6	2 663
CORR†	256	CORR	$(3N+2)N+1$	2.7	190
COVAR†	256	COVAR	$(3N+2)N$	2.6	219
FDTD†	2 048	Step1, 2, 3	6, 4, 6	58.4	3 887
GESUMMV†	4 096	GESUMMV	$8N+2$	64.8	5 043
GRAMSC.†	128	K3	$7N^2$	1.5	91
MVT†	4 096	K1, K2	$3N+1$	14.4	4 810
SYR2K†	256	SYR2K	$5N+1$	80.3	9 900
SYRK†	256	SYRK	$3N+2$	57	4 085
BP‡	262 144	K1, K2	16	28.6	800
CFD‡	0.2 M	Flux	83	51.7	1 626
HSPOT‡	1 024	HSPOT	3	14.1	817
NW‡	4 096	K1, K2	8 384	13.6	819
SRAD_V2‡	2 048	K1, K2	12	11.4	639
Average				**46.9**	**2 761**

† From Polybench/GPU [5], ‡From Rodinia V3.1 [6]

Table 4. Polybench/GPU and Rodinia benchmarks, specifications, and slowdowns of RDGC and GPGPU-Sim with respect to the executions performed on a GTX 970 GPU

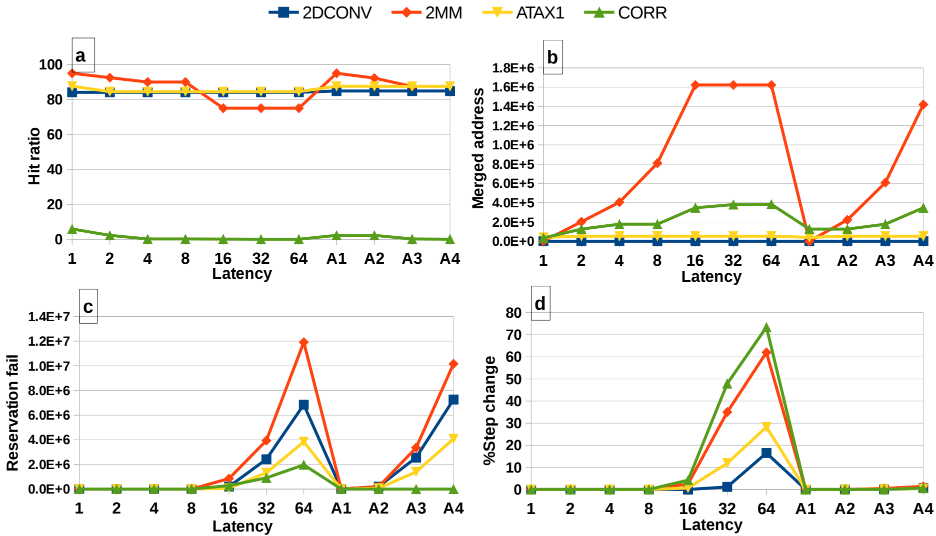

Figure 5. The effects of latency on different parameters in the model

5.2 RDGC Validation

RDGC is validated through comparing its outputs with the results provided by NVPROF for profiling the same workload on a Maxwell GTX 970 GPU and a Kepler GT 740M. Table 5 gives the parameter values used by RDGC. These values are selected based on the available NVIDIA documents [35] and the findings of previous studies [37]. However, some important cache parameters are neither reported by NVIDIA, nor discovered by the research community, e.g., L2 mapping function, the number of L1/L2 access ports, and L2 replacement policy. The analysis results of GTX 970 are shown in Figure 6 (hit ratios), Figure 7 (transaction counts). In addition, Figure 8 gives a comparison between the profiling results on a GT 740M and analysis results provided by RDGC.

Parameter	GTX 970	GT 740M
P	26[1]	2
L1 (¡S_1,K_1, B_1, Map., Repl.¿)	24 KB, 192, 32 B, XOR, LRU	–
L2 (¡S_2,K_2, B_2, Map., Repl.¿)	1792 KB , 8, 32 B, XOR, LRU	512 KB, 8, 32 B, XOR, LRU
Blocks to SM mapping	RR	RR

[1] GTX 970 has 13 SMs and each SM has two 24 KB L1 cache partitions, hence P and S_1 were set to 26 and 24 KB, respectively.

Table 5. The main configuration parameters of RDGC

5.2.1 The Physical Model Slowdowns

The physical model slowdowns were calculated through dividing their execution times measured on a system with Ubuntu 12.04 OS, Core i5 CPU, 4 GB of RAM, by the kernel execution times measured on a GTX 970 GPU (CUDA 7.0). All the kernel data transfer times are excluded. Further, the time overheads of the logical model were not included in the slowdown calculations. As shown in Table 4, the physical model had an average slowdown of 47K times, where 3DCONV had the highest slowdown (212 K times) as opposed to Gramschmidt with the lowest slowdown (1 504 times). It is worth mentioning that the performance of RDGC can be enhanced by employing some techniques such as parallel execution and statistical sampling methods [27]. The average slowdown of GPGPU-Sim measured 2 761 K (power simulation and visualizer was disabled). Therefore, RDGC (taking several minutes per application) is 59 times faster than the cycle-accurate simulation, while most of the applications take several hours to be simulated by GPGPU-Sim.

5.2.2 Discussion

According to the findings presented in Figures 6 to 8, the model has a fair accuracy in predicting the hit ratios and transaction counts. For the Maxwell GPU, The average absolute errors of $L1_R_Hit$, $L2_Hit$ and $L2_Hit_L1B$, were 3.72 %, 4.5 %, and 4.52 %, respectively. Further, the average error of $L1_R_Trans$, $L2_Trans$ and $L2_Trans_L1B$ were 15.0 %, 11.9 %, and 7.6 %, respectively. In the case of the

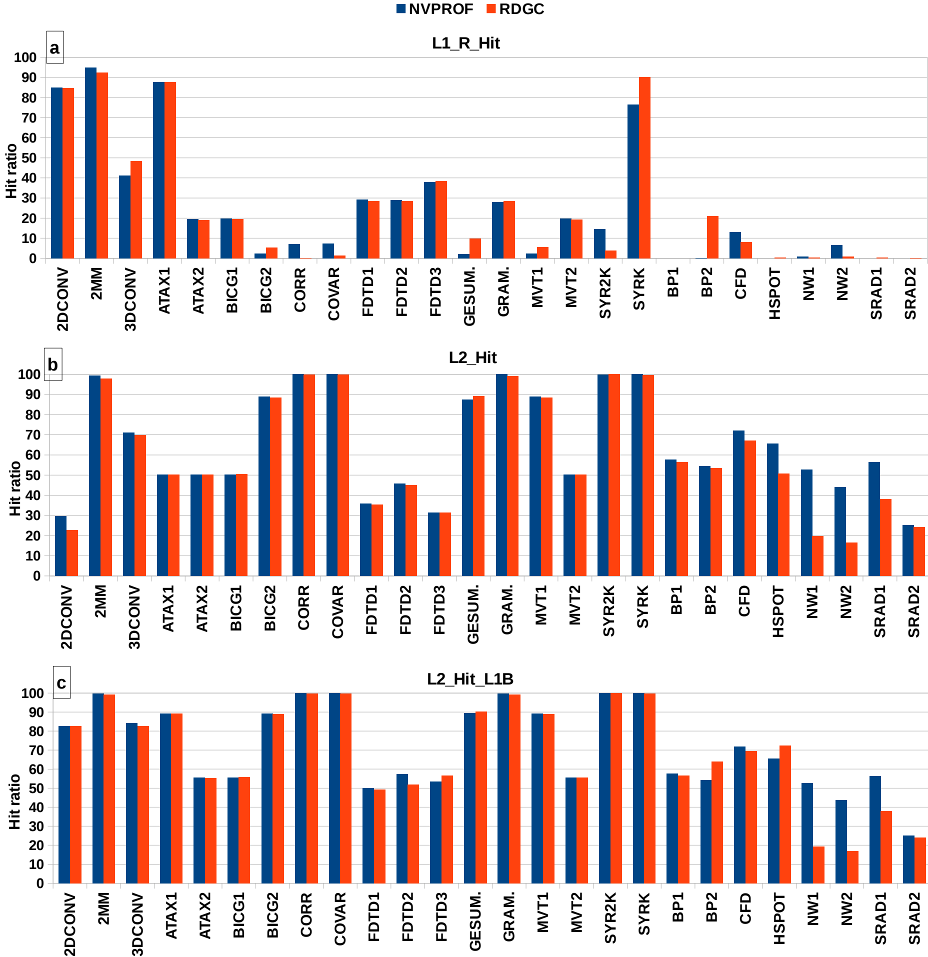

Figure 6. RDGC correlation with Maxwell GPU (GTX 970) (hit ratios)

Kepler GPU, the results has the average error of 5.4 % for *L2_Hit_L1B* and the average error of 5.6 % were observed for *L2_Trans_L1B*. Furthermore, GPGPU-Sim has an average error of 23.3 % and 11.7 % for *L1_R_Hit* and *L2_Hit*, respectively. Note that any error in *L1_R_Hit* may cause a high amount of error in *L2_Trans* (and *L2_Hit*). Moreover, since the size of the L1 cache is limited, *L1_R_Hit* is sensitive to L1 cache parameters. Moreover, in some kernels which extensively use shared memory, if the content of shared memory is spilled to the global memory, some transactions are generated at L1 and L2 caches to carry the spilled data. In our model, this phenomenon is ignored at L2 which can cause some error. Nevertheless, this phenomenon has only occurred in NW benchmark.

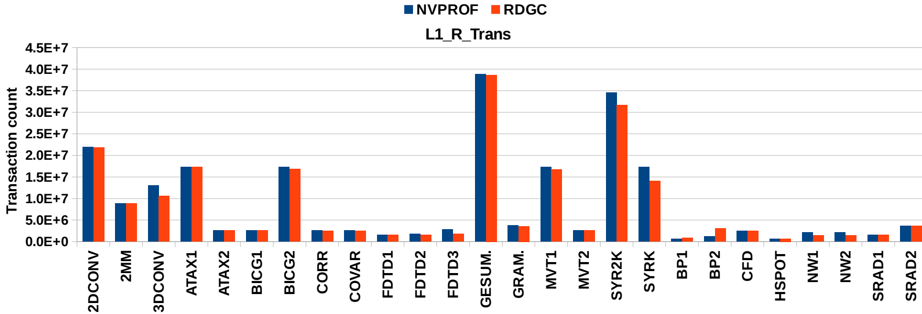

Figure 7. RDGC correlation with Maxwell GPU (GTX 970) (transaction counts)

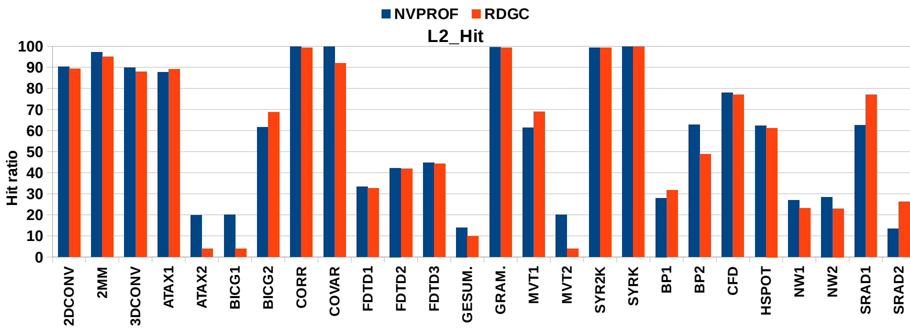

Figure 8. RDGC correlation with Kepler GPU (GT 740M)

Table 6 compares the RDGC model and the work of Nugteren et al. [8].

5.3 The Architectural Space Exploration of GPU Cache

In this section, different cache design parameters are explored. Only 2DCONV, 2MM, ATAX1 and CORR kernels were included in the evaluation. The selected kernels have diverse specifications in terms of their maximum number of per-thread accesses, grid dimensions, and block dimensions. The baseline GPU parameters applied for the simulations include 8 SMs, 32 KB L1 4-way set-associative in the form of a double-ported cache bank, 1 024 KB L2 8-way set-associative with four partitions and two 128 KB banks per partition, PRI mapping for L1 and L2, LRU replacement for L1 and L2, 128 B cache block size, and RR thread block to SM mapping policy. In total, 263 simulations were performed.

5.3.1 Analyzing the Effects of Cache Organizational Parameters

Cache size: The results of different cache sizes are shown in Figure 9. As can be seen, even small L1 caches result in large hit ratios in 2MM. In addition,

Specification	RDGC	Nugteren et al.
Coverage	L1 and L2	L1
Modeled GPU	Kepler (GT 740M), Maxwell (GTX 970)	Fermi (GTX 470)
Cache parallelism	multiple partitions, banks, ports	None
Mem. latency model	Adaptive	Probabilistic
Cache bypassing	Coarse	–
parameters	Hit ratio, Transaction count	Miss rates
Cache parameters	Capacity, associativity, block size	Capacity, associativity block size
Cache replacement	LRU, LFU, FIFO, Random	LRU
Mapping function	PRI, XOR, MOD, SMOD	Custom XOR
Block to SM mapping	RR, Partitioned, Random	RR

Table 6. Comparison of RDGC with the work of Nugterent et al. [8]

2DCONV highly benefits from increasing the L1 cache capacity. For instance, doubling the size of the 8 KB L1 cache led to increasing the hit ratio by 78 %. (Figure 9 a)).

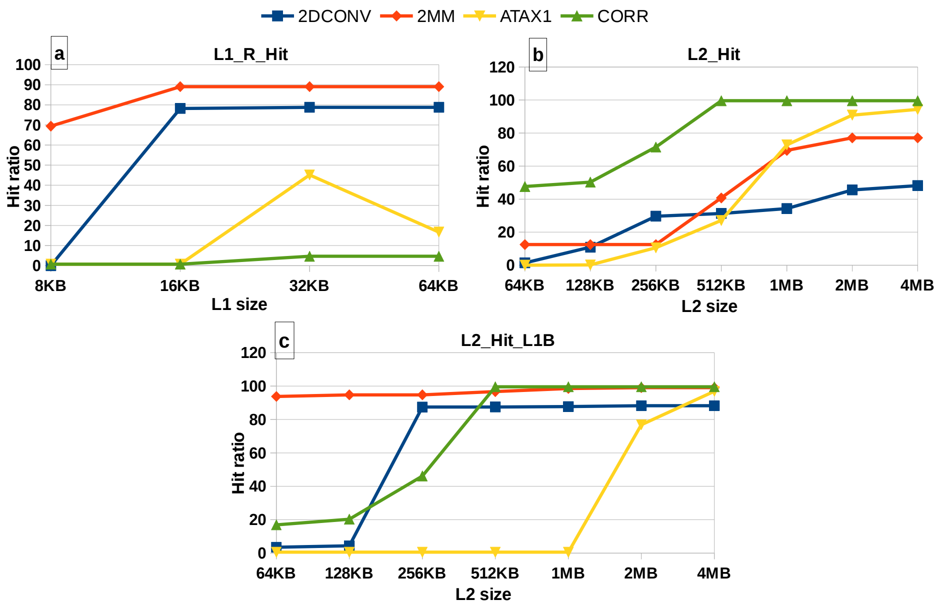

Figure 9. L1 and L2 performance for different cache sizes

Cache Mapping Function: In Figure 10, the *L1_R_Hit*, *L2_R_Hit*1 and *L2_R_Hit*2 metrics are presented for different cache mapping functions. As it can be observed, PRI functioned better than others, while the resultant hit

ratios significantly declined in several cases in MOD and shifted MOD mappings.

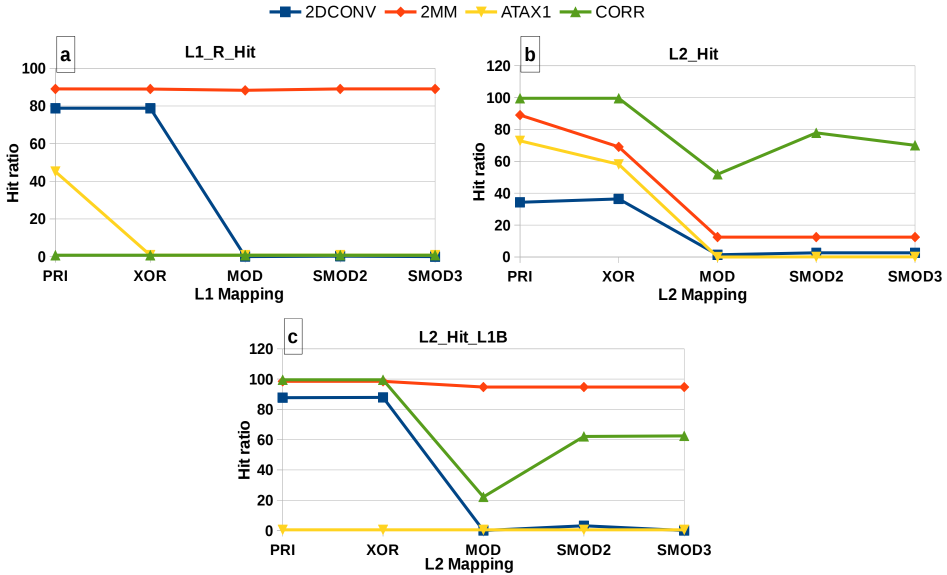

Figure 10. L1 and L2 performance for different cache mapping functions

Replacement Policy: In Figure 11, the metrics for different cache replacement policies are shown. For L1 cache, the performance of CORR was enhanced by LFU, whereas the performance of other three kernels was reduced. Further, ATAX1 achieved the best performance with random replacement in both L1 and L2 caches. Additionally, the performance of 2DCONV diminished as a result of employing LFU, but remained the same for other policies. On the other hand, 2MM showed less sensitivity to replacement policies than the other kernels. Overall, cache replacement policy is an important organizational parameter in cache memories, especially in L1 cache. Since no replacement policy functions the best all the time, employing the adaptive replacement policies is a promising approach.

Cache Block Size: In Figure 12, the resultant performance of different cache block sizes in L1 and L2 caches are illustrated. Except for ATAX1, the performance of L1 did not significantly change. Note that when L1 hit ratio is high, any small changes in L1 hit may result in radical changes in the transaction counts and hit ratios of L2 caches.

Cache Associativity: In Figure 13, the resultant metrics for different associativity (32 KB L1 and 1 MB L2) are shown. Figure 14 shows the RD profile for L2 cache including both read and write transactions. Note that the RD8+ in this

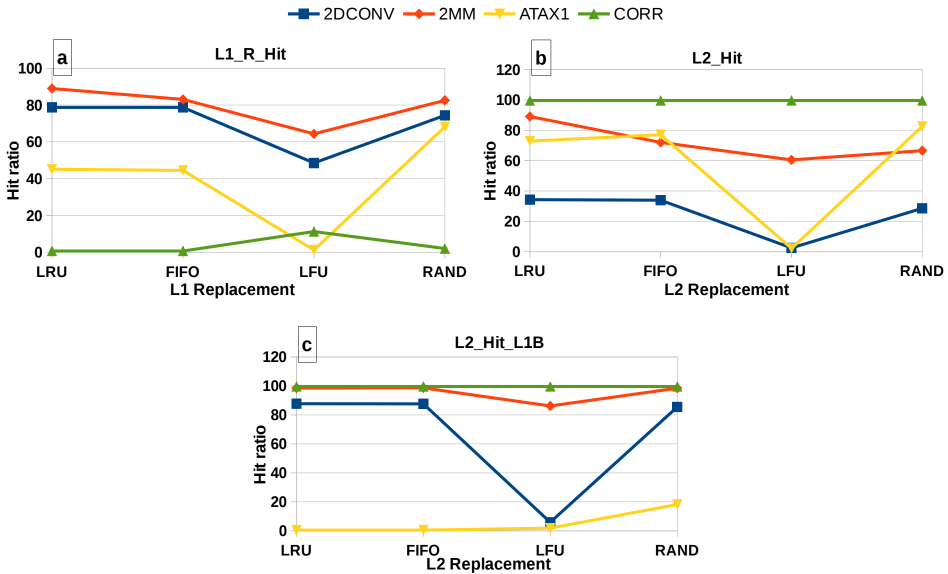

Figure 11. L1 and L2 performance for different cache replacement policies

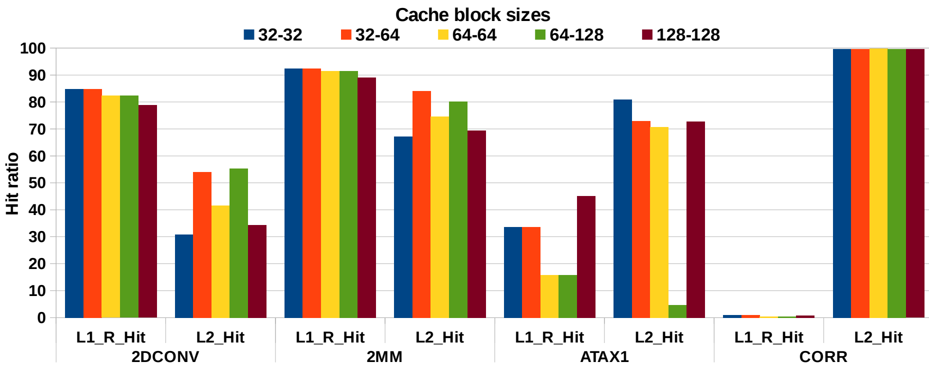

Figure 12. Cache performance for different cache block sizes

figure shows the missed Access. Moreover, RD profile is very helpful for performance analysis and characterization of application data reuse in many memory intensive GPU applications [28].

5.3.2 Blocks to SM Mapping

In Figure 15 the results of different CUDA thread blocks to SM mappings (see Table 2) are shown. Since CORR has only eight blocks, its results are not presented here. The results indicated that L1 cache performance was more sensitive to the

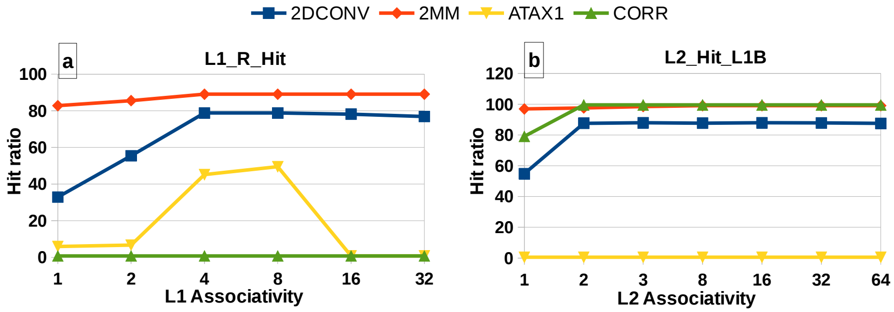

Figure 13. L1 and L2 performance for different cache associativity

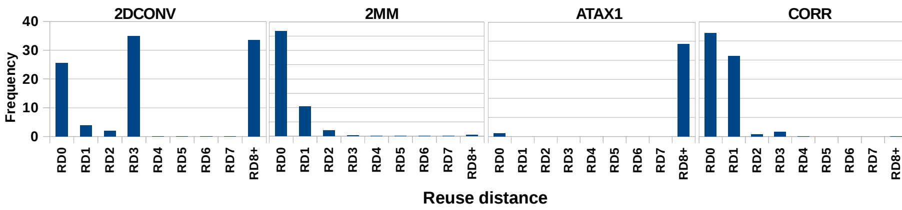

Figure 14. RD profile for L2 Cache (RWON)

blocks to SM mapping policies than L2. For example, ATAX1 achieved 65 % and 62.5 % of hit ratios for PART1 and PART2 and 45 % and 42 % of hit ratios for RR and Random block mapping policies, respectively.

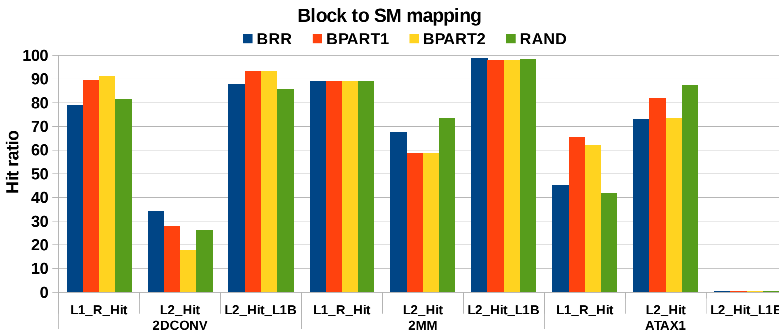

Figure 15. Cache performance for different thread blocks to SMs mapping policies

5.3.3 L2 Cache Parallelism Modeling

In this section, the performance of L2 cache (L1 disabled) for different cache parallelism levels are presented (see Figure 1 b)). Since GESUMMV has a considerable

number of transactions, it is included in this experiment. The two-level interleaving scheme was used for address mapping. In Figure 16, calculated MRU counters are presented. As it can be observed, cache parallelism changes the achieved reuse distance values, thus it is necessary to include the effects of cache parallelism in RD calculation. Even in 2DCONV and 2MM kernels that the hit ratios remained the same with different L2 parallelism levels, the reuse distance values were changed. These changes show that for bigger workload sizes or cache capacities, cache parallelism can alter the resultant hit ratios. Finally, Figure 17 shows the change in step counts as a function of L2 cache parallelism level.

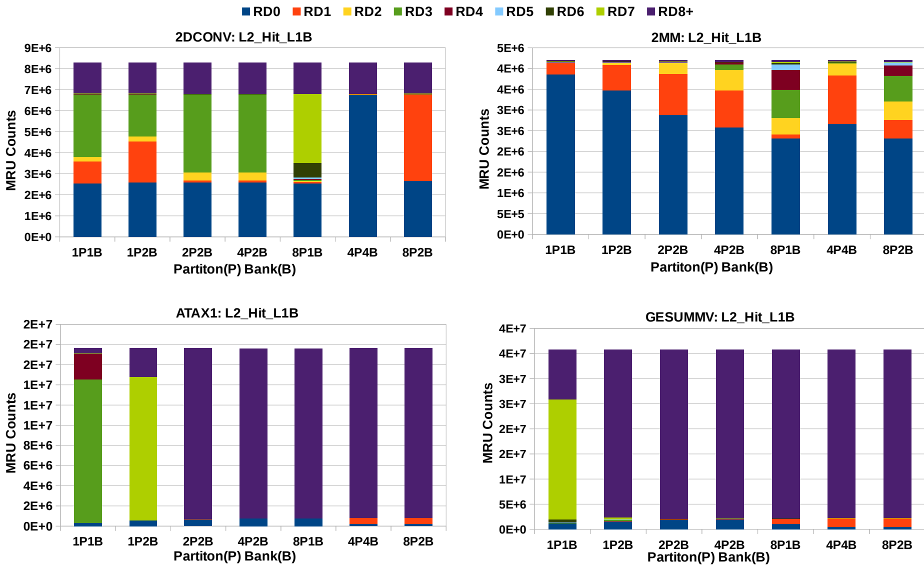

Figure 16. Reuse stack distances for different L2 parallelism levels (partitions (P), banks (B))

5.3.4 Discussion

While exploring the architectural space of cache memories through cycle-accurate simulation is extensively time-consuming, RDGC offers a more time-efficient approach to profile data locality and to model cache performance. However, RDGC is not a replacement for cycle-accurate simulation. Instead, it can be employed to narrow down the vast architectural space of GPU cache memories by analyzing different candidates for cache memory organization. RDGC can also be used by application developers to profile the data reuse. In addition, RDGC is agile to changes and can be modified to analyze caches in newer GPUs, without spending much effort.

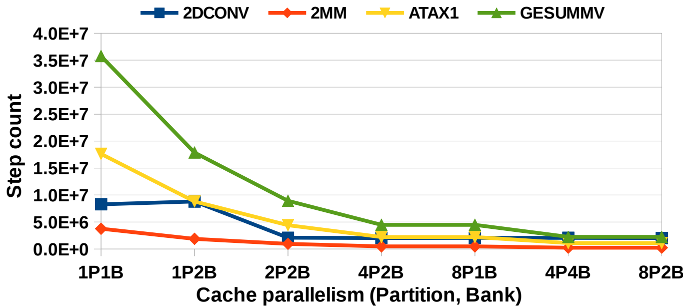

Figure 17. Step count trend for different L2 parallelism levels (partitions (P) and banks (B))

6 CONCLUSION

GPU architects and application developers need to analyze cache memory performance for different cache design parameters and different application configurations. Analyzing the performance of cache memories is a time-consuming task, especially for GPUs that execute threads in massive parallelism. This paper proposes a performance analysis approach, called RDGC, that applies reuse distance analysis to analyze the performance of GPU cache memory hierarchy. The evaluation results show that RDGC has fair performance and accuracy: 59 times speedup over GPGPU-Sim and an absolute error of 3.72 % and 4.5 % for L1 and L2 cache read hit ratios. Further, RDGC facilitates the architectural space exploration of GPU cache hierarchy. Different cache architectural parameters were modeled including: capacity, associativity, mapping (indexing), block size, replacement policy, and bypassing. In addition, the effects of cache parallelism (multi-bank and multi-port caches) were modeled. RDGC can be enhanced through including more advanced architectural specifications, i.e., adaptive replacement policies, advanced indexing functions, fine-grained access bypassing, warp scheduling algorithms. Further, RDGC can be adapted for modeling cache performance in multiple simultaneous kernel execution scenarios. In addition, RDGC can benefit from introducing more realistic latency models and inclusion of atomic instructions and cache coherency protocols.

REFERENCES

[1] Patterson, D.: The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. NVIDIA Whitepaper, Vol. 47, 2009.

[2] Jang, B.—Schaa, D.—Mistry, P.—Kaeli, D.: Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. IEEE Transactions on Parallel and Distributed Systems, Vol. 22, 2011, No. 1, pp. 105–118, doi: 10.1109/TPDS.2010.107.

[3] John, L. K.—Eeckhout, L.: Performance Evaluation and Benchmarking. CRC Press, 2005.

[4] Beyls, K.—D'Hollander, E. H.: Reuse Distance as a Metric for Cache Behavior. Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems, 2001, pp. 617–622.

[5] Pouchet, L.: Polybench/C: The Polyhedral Benchmark Suite. `http://www.cs.ucla.edu/~pouchet/software/polybench`, 2012.

[6] Che, S.—Boyer, M.—Meng, J.—Tarjan, D.—Sheaffer, J. W.—Lee, S.-H.—Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. 2009 IEEE International Symposium on Workload Characterization (IISWC 2009), 2009, pp. 44–54, doi: 10.1109/IISWC.2009.5306797.

[7] Tang, T.—Yang, X.—Lin, Y.: Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS '11), IEEE, 2011, pp. 623–634, doi: 10.1109/ICDCS.2011.16.

[8] Nugteren, C.—van den Braak, G.-J.—Corporaal, H.—Bal, H.: A Detailed GPU Cache Model Based on Reuse Distance Theory. 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2014, pp. 37–48, doi: 10.1109/HPCA.2014.6835955.

[9] Wu, M.-J.—Zhao, M.—Yeung, D.: Studying Multicore Processor Scaling via Reuse Distance Analysis. ACM SIGARCH Computer Architecture News – ICSA '13, ACM, Vol. 41, 2013, No. 3, pp. 499–510, doi: 10.1145/2508148.2485965.

[10] Cui, H.—Yi, Q.—Xue, J.—Wang, L.—Yang, Y.—Feng, X.: A Highly Parallel Reuse Distance Analysis Algorithm on GPUs. 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2012, pp. 1080–1092, doi: 10.1109/IPDPS.2012.100.

[11] Lee, S.—Ro, W. W.: Parallel GPU Architecture Simulation Framework Exploiting Architectural-Level Parallelism with Timing Error Prediction. IEEE Transactions on Computers, Vol. 65, 2016, No. 4, pp. 1253–1265, doi: 10.1109/TC.2015.2444848.

[12] Wu, M.-J.—Yeung, D.: Identifying Optimal Multicore Cache Hierarchies for Loop-Based Parallel Programs via Reuse Distance Analysis. Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '12), ACM, 2012, pp. 2–11, doi: 10.1145/2247684.2247687.

[13] Badamo, M.—Casarona, J.—Zhao, M.—Yeung, D.: Identifying Power-Efficient Multicore Cache Hierarchies via Reuse Distance Analysis. ACM Transactions on Computer Systems (TOCS), Vol. 34, 2016, No. 1, pp. 3–30, doi: 10.1145/2851503.

[14] Bakhoda, A.—Yuan, G. L.—Fung, W. W. L.—Wong, H.—Aamodt, T. M.: Analyzing CUDA Workloads Using a Detailed GPU Simulator. 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009), 2009, pp. 163–174, doi: 10.1109/ISPASS.2009.4919648.

[15] Ubal, R.—Jang, B.—Mistry, P.—Schaa, D.—Kaeli, D.: Multi2Sim: A Simulation Framework for CPU-GPU Computing. Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12), ACM, 2012, pp. 335–344, doi: 10.1145/2370816.2370865.

[16] Hong, S.—Kim, H.: An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. ACM SIGARCH Computer Architecture News, ACM, Vol. 37, 2009, No. 3, pp. 152–163, doi: 10.1145/1555815.1555775.

[17] Sim, J.—Dasgupta, A.—Kim, H.—Vuduc, R.: A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. ACM SIGPLAN Notices – PPOPP '12, ACM, Vol. 47, 2012, No. 8, pp. 11–22, doi: 10.1145/2370036.2145819.

[18] Baghsorkhi, S. S.—Gelado, I.—Delahaye, M.—Hwu, W. W.: Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors. ACM SIGPLAN Notices – PPOPP '12, ACM, Vol. 47, 2012, No. 8, pp. 23–34, doi: 10.1145/2370036.2145820.

[19] Huang, J.-C.—Lee, J. H.—Kim, H.—Lee, H.-H. S.: GPUMech: GPU Performance Modeling Technique Based on Interval Analysis. Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47), IEEE Computer Society, 2014, pp. 268–279, doi: 10.1109/MICRO.2014.59.

[20] Jia, H.—Zhang, Y.—Long, G.—Xu, J.—Yan, S.—Li, Y.: GPURoofline: A Model for Guiding Performance Optimizations on GPUs. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P. G. (Eds.): Euro-Par 2012 Parallel Processing. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7484, 2012, pp. 920–932, doi: 10.1007/978-3-642-32820-6_90.

[21] Dao, T. T.—Kim, J.—Seo, S.—Egger, B.—Lee, J.: A Performance Model for GPUs with Caches. IEEE Transactions on Parallel and Distributed Systems, Vol. 26, 2015, No. 7, pp. 1800–1813, doi: 10.1109/TPDS.2014.2333526.

[22] Kiani, M.—Rajabzadeh, A.: VLAG: A Very Fast Locality Approximation Model for GPU Kernels with Regular Access Patterns. 2017 7th International Conference on Computer and Knowledge Engineering (ICCKE 2017), October 26–27, 2017, Ferdowsi University of Mashhad, IEEE, 2017, pp. 260–265, doi: 10.1109/ICCKE.2017.8167887.

[23] Ding, C.—Chilimbi, T.: A Composable Model for Analyzing Locality of Multi-Threaded Programs. Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[24] Jiang, Y.—Zhang, E. Z.—Tian, K.—Shen, X.: Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors? In: Gupta, R. (Ed.): Compiler Construction (CC 2010). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6011, 2010, pp. 264–282, doi: 10.1007/978-3-642-11970-5_15.

[25] Schuff, D. L.—Parsons, B. S.—Pai, V. S.: Multicore-Aware Reuse Distance Analysis. 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Ph.D. Forum (IPDPSW), IEEE, 2010, pp. 1–8, doi: 10.1109/IPDPSW.2010.5470780.

[26] Wu, M.-J.—Yeung, D.: Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-Based Parallel Programs. 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11), IEEE, 2011, pp. 264–275, doi: 10.1109/PACT.2011.58.

[27] Schuff, D. L.—Kulkarni, M.—Pai, V. S.: Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10), IEEE, 2010, pp. 53–63, doi: 10.1145/1854273.1854286.

[28] Wang, D.—Xiao, W.: A Reuse Distance Based Performance Analysis on GPU L1 Data Cache. 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC), IEEE, 2016, pp. 1–8, doi: 10.1109/PCCC.2016.7820638.

[29] Mittal, S.: A Survey of Techniques for Managing and Leveraging Caches in GPUs. Journal of Circuits, Systems and Computers, Vol. 23, 2014, No. 8, Art. No. 1430002, doi: 10.1142/S0218126614300025.

[30] Rogers, T. G.—O'Connor, M.—Aamodt, T. M.: Cache-Conscious Wavefront Scheduling. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE, 2012, pp. 72–83, doi: 10.1109/MICRO.2012.16.

[31] Jog, A.—Kayiran, O.—Mishra, A. K.—Kandemir, M. T.—Mutlu, O.—Iyer, R.—Das, C. R.: Orchestrated Scheduling and Prefetching for GPGPUs. ACM SIGARCH Computer Architecture News – ICSA '13, ACM, Vol. 41, 2013, No. 3, pp. 332–343, doi: 10.1145/2508148.2485951.

[32] Xie, X.—Liang, Y.—Sun, G.—Chen, D.: An Efficient Compiler Framework for Cache Bypassing on GPUs. 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '13), IEEE, 2013, pp. 516–523, doi: 10.1109/ICCAD.2013.6691165.

[33] Kim, K. Y.—Baek, W.: Quantifying the Performance and Energy Efficiency of Advanced Cache Indexing for GPGPU Computing. Microprocessors and Microsystems, Vol. 43, 2016, pp. 81–94, doi: 10.1016/j.micpro.2016.01.003.

[34] CUDA NVIDIA. C Programming Guide Version 4.0. NVIDIA Corporation, 2011.

[35] NVIDIA: Maxwell Tuning Guide, 2017. Available at: `http://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html`, Accessed 18-February-2017.

[36] Li, C.—Song, S. L.—Dai, H.—Sidelnik, A.—Hari, S. K. S.—Zhou, H.: Locality-Driven Dynamic GPU Cache Bypassing. Proceedings of the 29th ACM International Conference on Supercomputing (ICS '15), ACM, 2015, pp. 67–77, doi: 10.1145/2751205.2751237.

[37] Mei, X.—Chu, X.: Dissecting GPU Memory Hierarchy Through Microbenchmarking. IEEE Transactions on Parallel and Distributed Systems, Vol. 28, 2017, No. 1, pp. 72–86, doi: 10.1109/TPDS.2016.2549523.

Mohsen Kiani is currently a Ph.D. student in computer engineering at Razi University, Kermanshah, Iran. His main research interests include computer architecture, many-core architectures, GPGPU, and performance modeling and analysis.

Amir Rajabzadeh received his B.Sc. degree in telecommunication engineering from Tehran University, Iran, in 1990 and his M.Sc. and Ph.D. degrees in computer engineering from Sharif University of Technology, Iran, in 1999 and 2005, respectively. He was a visiting researcher in the Embedded Systems Laboratory, University of Leicester, UK in summer 2005 and in the CARG Group, Ottawa University, Canada in 2012–2013. He has been working as Assistant Professor of computer engineering at Razi University, Kermanshah, Iran since 2005. He was the Head of the Computer Engineering Department (2005–2008) and the Education and Research Director of the Engineering Faculty (2008–2010) at Razi University. He has authored several journal papers and other refereed publications. His main areas of interests are computer architecture, high performance computing and fault-tolerant systems design. He has earned one world, six international, and five national awards in robotic competition, and one national award in mobile computing.