Efficient Cache Performance Modeling in GPUs Using Reuse Distance Analysis

MOHSEN KIANI and AMIR RAJABZADEH, Razi University, Iran

Reuse distance analysis (RDA) is a popular method for calculating locality profiles and modeling cache performance. The present article proposes a framework to apply the RDA algorithm to obtain reuse distance profiles in graphics processing unit (GPU) kernels. To study the implications of hardware-related parameters in RDA, two RDA algorithms were employed, including a high-level cache-independent RDA algorithm, called HLRDA, and a detailed RDA algorithm, called DRDA. DRDA models the effects of reservation fails in cache blocks and miss status holding registers to provide accurate cache-related performance metrics. In this case, the reuse profiles are cache-specific. In a selection of GPU kernels, DRDA obtained the L1 miss-rate breakdowns with an average error of 3.86% and outperformed the state-of-the-art RDA in terms of accuracy. In terms of performance, DRDA is 246,000× slower than the real GPU executions and 11× faster than GPGPU-Sim. HLRDA ignores the cache-related parameters and its obtained reuse profiles are general, which can be used to calculate miss rates in all cache sizes. Moreover, the average error incurred by HLRDA was 16.9%.

CCS Concepts: • **General and reference** → *Performance*; • **Computer systems organization** → *Single instruction, multiple data*; *Heterogeneous (hybrid) systems*;

Additional Key Words and Phrases: GPU, miss rate, reuse distance analysis, cache memory

ACM Reference format:

Mohsen Kiani and Amir Rajabzadeh. 2018. Efficient Cache Performance Modeling in GPUs Using Reuse Distance Analysis. *ACM Trans. Archit. Code Optim.* 15, 4, Article 58 (December 2018), 24 pages.
<https://doi.org/10.1145/3291051>

1 INTRODUCTION

Graphics processing units (GPUs) are one of the most common computing resources used in modern high-performance computing systems. Many general-purpose applications with high degrees of parallelism are ported to GPUs to enhance their performance. A considerable portion of GPU applications are memory-intensive [18]. Consequently, memory performance has emerged as a major performance bottleneck in GPUs. However, GPU memory systems have extensively been studied and many cache- and memory-related designs and techniques have been proposed by researchers to boost the performance of memory systems. Due to the substantial effects of cache memories on the overall memory performance, the studies have mainly been focused on cache memories, some of which include locality-aware warp scheduling [40], cache bypassing [23], cache indexing [22], cache management [20], and memory management schemes [2].

Authors' addresses: M. Kiani and A. Rajabzadeh, Department of Computer Engineering and Information Technology, Razi University, Taghe-Bostan, Kermanshah, Iran, 6714414971; emails: {kiani.mohsen, rajabzadeh}@razi.ac.ir.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.

1544-3566/2018/12-ART58

<https://doi.org/10.1145/3291051>

Modeling the performance of GPU cache memories is an important requirement for hardware architects, compiler designers, and application developers. To this end, many researchers have been working on the development of design/analysis methods for GPU memory systems [4, 27]. However, this field still faces serious challenges. In this regard, providing accurate, detailed, and insightful performance-related information within reasonable time budgets is very challenging.

Hardware architects typically rely on cycle-accurate simulation when experimenting with their cache or memory system designs. However, cycle-accurate simulators usually result in significant slowdowns with respect to real executions. Thus, given the numerous cache organizational parameters, this method is very time-consuming and is less likely to explore all the candidate designs. Consequently, many researchers have reported their results for a limited number of simulated instructions toward decreasing the simulation time. However, such incomplete simulations can result in misleading evaluations. Another problem with the cycle-accurate simulations is that, despite providing detailed information, a single simulation may fail to provide enough insight into the behavior of a particular hardware module. Further, the application of cycle-accurate simulations is very confined.

Compiler designers and application developers also require effective methods to determine how efficiently an application performs on its target platform. Then, proper optimization techniques can be applied to the application. On this ground, reuse distance analysis (RDA) [5] can be used to quantify the data reuse in GPU applications. In RDA, a trace of memory references (at cache block granularity) is analyzed to calculate the reuse distance (RD) of all accesses in the trace. The RD value of a reference accessing a given data, d , is equal to the number of unique data other than d referenced between the access and the last previous access to the same data, d . RDA is appealing due to the provision of hardware-independent locality histograms, called RD profiles. An RD profile illustrates the occurrence frequency of each RD value, which can be utilized to analyze cache performance and to calculate miss-rate breakdowns (e.g., cold, capacity, and conflicting misses). Providing the miss-rate breakdowns is one of the most salient benefits of an RDA algorithm over other simulation methods. In addition, the RD profile obtained for an application can be used to calculate the miss rate of any arbitrary cache size, provided that the cache is organized as a fully associative least-recently used (LRU) cache.

Recently, RDA has been employed as a basis to estimate the total memory access time and to approximate the processor's performance and power [3] in multicore CPUs. Additionally, RDA is used for problem and processor scaling analysis in large-scale chip multiprocessors (LCMPs) [39]. Accordingly, RDA is not solely limited to miss-rate calculation and can be used as a basis to estimate other performance-related parameters.

Two major limitations can restrict how RDA is used. The first limitation is that the detailed hardware parameters are not modeled in RDA and the second is that calculating the RD profile of an application is time-consuming. As for the former, the basic RDA algorithm is a hardware-independent algorithm whereby a memory reference sequence is analyzed and the hardware-related parameters, including memory latency and thread-level parallelism, are ignored. Accordingly, the basic RDA algorithm fails to provide accurate results for specific hardware designs. Hence, when considering the detailed performance modeling, the hardware-level parameters should be included in the RDA algorithm. Furthermore, because modern processors are multi- or many-core processors, proper modifications should be applied to include the effects of parallelism in RDA. Including the hardware-related factors degrades the generality of RDA (i.e., its hardware dependence). Therefore, the outputs obtained for a specific hardware configuration cannot be used for other processors. In other words, a tradeoff exists between hardware independence and the accuracy of RDA algorithm.

As for the latter, the excessive time-complexity of RDA, even tree-based RDA implementations, result in $O(T \log(N))$ time complexity (where T and N stand for the total number of memory

references and unique referenced data, respectively). This problem is aggravated in cases where RDA is extended for GPU applications, in which typically hundreds or thousands of concurrent threads are included in the analysis. Therefore, some appropriate techniques should be employed to mitigate the complexity of RDA calculations.

Although RDA has been adapted for multicore CPUs by many researchers [37, 39], only a few contributions have been devoted to the extension of RDA for GPUs. Further, only the L1 cache memories were analyzed in the existing contributions [27, 35]. As a result, the adaptation of RDA for GPUs deserves more attention. For example, Nugteren et al. [27] proposed a detailed RDA model wherein memory latency as well as reservation fails in miss status holding registers (MSHRs) were modeled. However, the model ignores the L2 cache effects. Moreover, neither writing memory references nor some other important performance parameters, e.g., block reservation fails (*BRFs*), were modeled. A *BRF* occurs when all the blocks on a cache set are already reserved by the ongoing misses and a new access, which missed on the set, attempts to reserve a block in the set. To alleviate the computing complexity, the authors implemented their RDA algorithm based on Bennett and Kruskal's algorithm, which requires $O(T \log(T))$ time [1]. In addition, the simulations were performed for a limited number of cores and threads. However, generalizing the results of a limited number of threads, without properly selecting the representative threads, may result in deviations in the estimated performance, especially in cases in which the threads exhibit distinct per-thread memory access patterns.

In the present study, we propose a framework for the application of RDA to cache performance modeling in GPUs. Further, we evaluated two RDA algorithms including a hardware-independent and a detailed algorithm. The former ignores micro-architectural parameters. Thus, the acquired RD profile can be used for analyzing all cache sizes. The detailed RDA algorithm provides accurate performance metrics for a given cache configuration. As a result, the RD profile obtained using this method is only applicable to the simulated cache configuration. Both L1 and L2 cache memories are analyzed using the detailed RDA and thereby making it capable of capturing the nonlinear behaviors in the cache memories. Our evaluation results indicate that ignoring the effects of hardware parameters can result in significant errors in miss-rate estimation.

More specifically, the proposed framework (presented in Section 3) models the GPU's massive thread parallelism to obtain proper cache reference sequences within each streaming multiprocessor (SM). Moreover, in the detailed RDA algorithm, which is an extension to the state-of-the-art algorithm, various cache performance parameters are modeled, including backing stores latency, MSHRs, and *BRFs*. In contrast to other approaches in which the per-SM L1 cache memories are analyzed one by one [27], in our proposed model, all L1 cache memories are simultaneously analyzed step by step to better capture the L1-L2 cache interactions. The simultaneous analysis of the L1 and L2 cache memories enables more accurate cache latency modeling: the latency of a missed memory reference at the L1 can be estimated more accurately, if the outcome (hit/miss) of the reference in the L2 is defined. Moreover, this approach can contribute to modeling the coherency in L1 and atomic instructions in L2 cache. However, we leave these subjects for future works. The high-level RDA method is used to examine the extent to which ignoring the detailed parameters, to achieve hardware-independence, affects accuracy. When the high-level RDA analysis is of interest, it can be replaced with a sampling method, which largely reduces calculation time. To this end, we further examine a statistical sampling method [12] (presented in Section 5) and report our findings. We evaluated our proposed framework through comparing the outputs obtained for several memory-intensive GPU kernels with the performance parameters of real executions on Maxwell and Kepler GPUs as well as a state-of-the-art RDA algorithm (presented in Section 6) and a GPU cycle-accurate simulator.

Fig. 1. Overview of the memory hierarchy in NVIDIA GPUs.

2 BACKGROUND

In the present study, NVIDIA GPUs and their terminology are considered. In this section, first, a brief background on NVIDIA GPUs and their cache memory organization is presented, and then the basic RDA algorithm is explained.

2.1 NVIDIA GPUs and Their Memory Hierarchy

NVIDIA GPUs consist of several single instruction multiple thread (SIMT) processing units called streaming multiprocessors (SMs), where each SM embodies both processing elements and memory spaces. The per-SM memory spaces include a register file, a software-managed cache memory (called shared memory), and several hardware-managed cache memories including the L1 data and instruction caches. In some GPUs, e.g., Maxwell generation, more than one L1 partition is implemented on the SM. The L1 cache memories only cache read references and follow a write-evict policy, i.e., in the case of a write hit, the block is evicted from the cache. Further, some GPUs can optionally enable or disable the L1 caches [29]. To handle the huge memory traffics generated by the SMs, a second level (L2) cache memory is placed between the SMs and main memory of the GPU. The L2 cache consists of several cache partitions. Figure 1 presents an overview of the GPU cache hierarchy.

Miss status holding registers (MSHRs) are employed to handle the outstanding missed memory references. When a cache reference misses on the cache, first, all the outstanding memory references are checked to see if there is a previous pending reference to the same cache block. If such reference exists in an MSHR, then the new missed reference will merge into it (such an event is also called a pending hit). However, each MSHR can only manage a limited number of references. Thus, the number of allowed merges per MSHR is limited. If an MSHR currently contains the referenced address while containing the maximum accesses it can handle, or if no reference to the requested address exists in any MSHR, then a new MSHR should be assigned to the missed reference. In this case, since the total number of MSHRs is also limited (typically to 32), if no free MSHR is available, an MSHR reservation fail (denoted by *MRF*) occurs and the issuing pipeline stalls until an MSHR becomes available. Another limiting event, which incurs the stall cycles, is a block reservation fail (denoted by *BRF*). When there are K outstanding memory references to the same cache set (K is the cache associativity), which means that all the cache blocks of that set have been reserved by those pending misses, no new reserve can be accepted at the set. As a result, a new miss at that set fails to reserve a block [8, 14], which results in a *BRF*. In a memory-intensive kernel, MSHR or *BRFs* can impose serious stall cycles. The occurrence of such stall cycles can result in non-linear behaviors and affect the cache and memory performance. This necessitates that such hardware-level effects be modeled in an accurate cache model.

Fig. 2. RDA applied to a sample reference sequence and the acquired RD profile.

2.2 Reuse Distance Analysis

Reuse distance analysis (RDA) [5, 24] is a popular method for analyzing the data reuse in a memory reference sequence. In RDA, a parameter, called reuse distance (RD) (a.k.a. stack distance) is calculated for each memory reference. RD of a memory reference (r) to a given data (d), is measured as the number of unique memory references to other data occurring between r and the previous reference to d . The RD value of the first reference to an address is considered to be ∞ , which represents a cold miss. Once the RD values are calculated for all the memory references, a histogram can be plotted, in which each bin represents the frequency of an RD value. The reuse histogram is also called RD profile. With RD profile, the hit rate of a fully associative LRU cache memory of S blocks is the ratio of the number of memory references with $RD < S$ to the total number of references. Therefore, by performing RDA once, the miss rates of all cache sizes can be calculated.

RDA is a hardware-independent analysis that is applied to a given sequence of memory reference to acquire its RD profile. Additionally, RDA can be used for cache performance analysis, and the cache miss-rate breakdowns can be calculated from the RD profiles. Thus, one significant benefit of RDA is that it provides the cold and capacity miss breakdowns. Figure 2 represents how RDA is applied to a sample reference sequence to acquire its RD profile. For a cache of four blocks, the miss rate equals 50% with no capacity miss, while for a cache of two blocks, the miss rate is equal to 75%.

3 PROPOSED FRAMEWORK FOR CACHE PERFORMANCE MODELING

We extend RDA for GPUs and employ it for cache performance modeling. Because the basic RDA algorithm operates on a serialized cache reference sequence whereas GPUs are parallel processors, a framework is developed to obtain the serialized references to the cache memories in a GPU. The overview of the framework is depicted in Figure 3. It consists of three parts as follows:

3.1 Trace Extraction

First, the necessary memory access information is extracted from the kernel and stored in a file called memory trace (or trace). Since only the raw (i.e., the application-related) information is extracted in this step, the trace is GPU-independent. By doing so, the trace can be used to simulate any specific GPU (done in the next step of the framework). For each (warp-level) instruction within a CUDA block, a unique instruction identifier (i.e., program counter) is assigned. Further, the index of the block to which the warp belongs, an intra-block warp index, the number of active threads in the warp, and the requested addresses of the threads are recorded. Moreover, a Boolean variable is assigned to each instruction to define whether it is a global load or store. In our work, the necessary information was extracted through manually instrumenting the kernels' source codes. Kernels are then executed and their traces, containing the mentioned information, are acquired. An alternative to our approach is using compilers or emulators [13]. It should be noted that the trace extraction step is performed once.

Fig. 3. The proposed framework for the analysis of GPU L1 and L2 cache memories.

In our method, an intra-warp (control flow) divergence is handled at the trace extraction step. If a warp diverges, then each of its branches stores its memory instructions in the trace, each assigned with a unique instruction identifier until the threads of the diverged warp re-converge. It should be noted that the number of active threads in each instruction is also recorded.

3.2 Reference Sequence Generation

In this step, an execution-driven approach is adopted, which is responsible for modeling a specific GPU. All the important GPU-related parameters are used to obtain the serialized cache references. In the context of the present study, modeling a GPU is realized through generating the cache reference sequences, so that the thread scheduling of the specified GPU can be simulated. Modeling the thread scheduling includes mapping the CUDA blocks to the SMs, selecting the in-flight blocks, scheduling the in-flight warps, and serializing the references generated by the executing warps.

The block to SM mapper shown in Figure 3 maps the blocks to the SMs under the defined mapping policy. In this study, we consider three mapping schemes as follows:

- (1) Round-Robin block mapping (BRR): The blocks are assigned to the SMs under the round-robin policy. Thus, the adjacent blocks are assigned to different SMs. This method can degrade the exploited inter-block locality.
- (2) Random block mapping (BRND): Blocks are distributed randomly among the SMs.
- (3) Block partitioning (BPRT): Blocks are partitioned among the SMs so that each SM is assigned with a contiguous partition of adjacent blocks.

The next step toward the generation of reference sequences to the cache memories is defining the number of in-flight blocks in each SM. According to the CUDA programming model, all the blocks can be executed in parallel. However, the physical execution model is not the same as the programming model. The SM's resource limitations restrict the number of in-flight blocks. The number of in-flight blocks is defined based on two hardware-related parameters, including the maximum number of in-flight blocks and the maximum number of in-flight warps. Depending on the kernel's grid and block dimensions, one of the mentioned parameters can limit the number of in-flight blocks. The mentioned parameters and the number of SMs are configurable in the framework.

Once the number of in-flight blocks is defined, the in-flight blocks, which are selected based on indexes of the blocks, are further divided into warps and scheduled by the schedulers. As

illustrated in Figure 3, the number of per-SM schedulers is configurable where the schedulers perform the warps in parallel under a specific scheduling policy. The warp scheduling policy determines the order in which the in-flight warps are executed, thereby affecting the exploited locality and the resultant cache performance. In the present study, three warp scheduling policies are modeled. Before describing them, the mapping of in-flight warps to schedulers is explained.

In all of the modeled scheduling policies, the mapping is as follows. Let nW and nS denote the number of in-flight warps and per-SM schedulers, respectively. Warp i , $i \in [0, \dots, nW - 1]$ is assigned to scheduler j , $j \in [0, \dots, nS - 1]$ if $(i \% nS) = j$. NVIDIA GPUs use a similar method. In this work, all of the used scheduling policies simulate a barrel processing model [26], whereby all of the accesses of the selected warp are analyzed prior to switching to the next warp. The modeled warp scheduling policies are as follows:

- (1) Round-Robin warp scheduling (WRR): Starting from the first warp assigned to the scheduler, one instruction of each warp is performed, and then the scheduler selects the next warp and continues the execution. After one round, the execution begins from the next instruction of the first warp. Note that analyzing one instruction includes all of the transactions generated by the instruction. In this method, the warp schedulers can proceed independently.
- (2) In-line Round-Robin warp scheduling (WILRR): It is similar to the above policy, with the exception that all in-flight warps have the same value of program counter. Thus, if one scheduler finishes one round of execution earlier than other schedulers, it waits for other schedulers to finish the current instruction. This method improves the exploited inter-warp data locality.
- (3) Greedy-Then-Oldest warp scheduling (WGTO): A warp is selected and executed until it encounters a stall (a reservation fail or an L2 miss). When a scheduler puts a warp aside due to a stall, it picks the oldest ready warp among the in-flight warps. It should be noted that the full instruction execution stream is required to properly model this scheduling policy, while here only the memory instructions stream is available. Nevertheless, the modeled policy has a similar behavior to the GTO policy, since the order of warp execution is altered due to a stall.

For each selected warp, all of its thread accesses are coalesced to form accesses at cache block size granularity and then placed into a queue denoted by $SchQi$ (for scheduler i), as shown in Figure 3. Since more than one scheduler can execute warps in parallel, the generated references by the schedulers should be serialized. This is done by putting one reference of each $SchQ$ into a serial queue, denoted by $RefQ$, which contains access sequence to the L1 cache, i.e., the trace. Hence, the thread-level parallelism in GPUs is translated to cache reference sequences.

As presented in the evaluations, our framework is tested on two GPU generations. However, since NVIDIA GPUs have similar cache organizations, the framework can be utilized to analyze other GPU generations. Maxwell GPUs have four warp schedulers and two L1 data caches per SM (each cache is dedicated to two schedulers). Further, Pascal GPUs have two schedulers and one L1 data cache per SM whereas the new Volta V100 generation has four schedulers and one L1 data cache per SM [28].

3.2.1 Reuse Distance Analysis (RDA). The cache reference sequences obtained in the prior step are then analyzed through applying a trace-driven simulation, e.g., the RDA method. Any other trace-driven simulation can also be utilized to analyze the GPU cache memories. Here, two types of RDA methods can be employed. One choice is utilizing a high-level RDA, in which no hardware-related parameter is modeled. Therefore, the obtained RD profile is valid for analyzing fully

associative caches of arbitrary size. The second choice is utilizing a detailed RDA method that obtains precise results for a given cache configuration. However, if a detailed RDA model is applied, the acquired RD profiles become hardware-dependent. For example, when the effects of MSHRs are included in the RDA, the acquired RD profile is accurate only for the simulated cache. The main benefit of the detailed RDA with respect to a simple trace-driven simulation is that it generates miss-rate breakdowns. In this work, we examine both high-level and detailed RDA methods.

In our methodology, the detailed RDA method is concurrently applied to all of the L1 cache access sequences. Starting from the first SM, one coalesced reference is retrieved from each *RefQ* and analyzed. After analyzing the memory references at the L1 level, the L2 analysis is conducted (on the references that missed the L1 caches). The outcome is used to define the proper latency of updating the states of the L1 caches (see Section 4.4). After one turn of analysis, the states of the L1 and L2 caches are updated properly by filling the cache with the arrived data from the backing stores. Then, the analysis continues with the next memory references in the *RefQs*.

Although one simulation is enough to calculate the miss rates of all cache sizes, even a high-level RDA algorithm is GPU specific. This is because a variety of hardware-related parameters are modeled to obtain the reference sequences. It means that the sequences can be used to analyze the performance of cache memories of different sizes and various MSHR counts. However, to model a different GPU, the second step of the framework should be performed to obtain the reference sequences.

If obtaining hardware-independent RD profiles is desired, then the sampling methods can be used instead of an RDA to alleviate its calculation time complexity. StatStack [12] is an example of hardware-independent sampling method that requires sparse samples of reuse time distance to estimate the RD profile (the reuse time distance counts the total number of blocks touched between two consecutive references to the same block). This method can extensively accelerate the RD profile calculation at the cost of lower accuracy. We augmented the reference sequence generation step to record sparse reuse time samples from each reference sequence and then employed the StatStack method to estimate the miss rates (see Section 5). In the next section, we explain the used RDA methods.

4 REUSE DISTANCE ANALYSIS AND CACHE PERFORMANCE MODELING

The main aim of the RDA is capturing locality in a reference sequence and converting it to cache performance metrics. The basic RDA is hardware independent. However, the effects of different hardware-related parameters can be included in RDA to enhance accuracy. The downside of a hardware-related RDA is that the RD profile achieved through a single simulation is not anymore applicable to analyze all cache sizes. In the present study, we employed two RDA methods. In one method, called high-level RDA (HLRDA), no detailed hardware parameter is modeled. In the second RDA method, called detailed RDA (DRDA), several performance-influential parameters are included. The used RDA methods are explained below.

4.1 High-level RDA Method

The high-level RDA (HLRDA) method employs the traditional RDA (see Section 2.2) to attain the RD profiles. HLRDA generates full RD profiles, which can be used to estimate the miss rates of fully associative LRU cache of any size. The implementation method of HLRDA is explained in Section 4.5.

4.2 Detailed RDA Method

The detailed RDA (DRDA) method models several performance-influential parameters. The modeled parameters include cache associativity, backing store accessing latency, and MSHRs.

Table 1. RDA Applied to a Sample Access Sequence to a Fully Associative Cache with Two Blocks and Backing Store Latency of Three

Step	1	2	3	4	5	6	7
Ref. sequence	a	b	c	a	c	d	b
RD	∞	∞	∞	0	∞	∞	1
State	m	m	m	h	m	m	h
Updated block	-	-	-	a	b	c	-

h and m represent a hit and a miss, respectively.

Table 2. RDA Applied to a Sample Access Sequence to a Fully Associative Cache with Two Blocks and Two MSHRs, and Backing Store Latency of Three

Step	1	2	3	4	5	6	7
Ref. sequence	a	b	c	c	a	c	d
RD	∞	∞	-	∞	0	∞	∞
State	m	m	MRF	m	h	$merge$	m
#res. MSHRs	$\uparrow 1$	$\uparrow 2$	2	$\downarrow \uparrow 2$	$\downarrow 1$	1	$\downarrow \uparrow 1$
Updated block	-	-	-	a	b	-	c

h and m represent a hit and a miss; $merge$ represents an MSHR merge, and MRF represents an MSHR reservation fail.

4.2.1 Modeling Backing Store Latency. The latency of transferring the data requested by a missed access depends on the organization of the memory system. This latency can alter the calculated reuse distances. Nugteren et al. [27] modeled the effects of the backing store latency. The idea is best explained using an example. Table 1 shows how RD values are calculated when the backing store latency is modeled. The second row shows the referenced data (at cache block granularity), and the two subsequent rows show the calculated RD values and the result of analyzing the accesses, respectively. The RD values are calculated based on the state of the cache in the respective step. Further, the state is either a hit (denoted by h) or a miss (denoted by m). When the reuse distance is less than the number of cache blocks, the access is a hit, otherwise, it is miss. The last row in Table 1 shows the step in which the cache is updated. The latency is equal to three. When the backing store latency is a constant value l , a missed reference at step s updates the cache at the end of step $s + l - 1$, and it is available in step $s + l$. For instance, the access to block a in step 1 causes a miss and the requested block arrives three steps later and is available from step 4 forward. Assigning the same constant latency to all of the missed accesses fails to simulate the memory latency in real GPUs. In Section 4.4, a proper model is developed to estimate the latency of each missed access.

4.2.2 Modeling MSHRs. An access reserves an MSHR on a miss to handle the requested data transfer and to update the cache when the data arrives. The number of MSHRs is limited. As a result, an access may fail to reserve an MSHR on a miss. This event is called MSHR reservation fail, denoted by MRF . Further, before reserving an MSHR, the missed access checks other MSHRs for an ongoing miss to the same data. If such MSHR exists, then instead of reserving a new MSHR, the access will merge into the existing access. This event is denoted by $merge$. However, the number of merged accesses per MSHR is also limited. When an MRF occurs, the cache cannot serve any new access and the issuing pipeline stalls. MSHRs largely affect cache performance [27]. Thus, if the RDA is intended to analyze the cache performance, the MSHRs should be modeled.

Table 2 shows an example for calculating the reuse distances of a sample access sequence when MSHRs are modeled. Here, there are two MSHRs. In the fourth row, the state can be a hit (denoted

Table 3. RDA for a Sample Access Sequence to a Two-way Set-associative Cache with Four Blocks and Infinite MSHRs and Backing Store Latency of Three (Figure 4)

Step	1	2	3	4	5	6	7
Ref. sequence	<i>a</i>	<i>c</i>	<i>e</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>c</i>
Cache set (S)	0	0	0	0	1	0	1
#res. Blocks (S0,S1)	↑1,0	↑2,0	2,0	↓↑2,0	↓1, ↑1	1,1	↓0, ↑2
RD	∞	∞	-	∞	0	1	∞
State	<i>m</i>	<i>m</i>	<i>BRF</i>	<i>m</i>	<i>m</i>	<i>h</i>	<i>m</i>
Updated block	-	-	-	<i>a</i>	<i>c</i>	-	<i>e</i>

h, *m*, and *BRF* represent a hit, a miss, and a block reservation fail, respectively.

Fig. 4. Two-way set-associative cache considered in the RDA method shown in Table 3.

by *h*), a miss (denoted by *m*), an *MRF*, or a *merge*. The fifth row shows the number of reserved MSHRs. An upward arrow in this row shows an MSHR reserve and a downward arrow represents an MSHR release. A downward arrow in step *s* shows that the MSHR is released at the end of step *s* - 1 and it is available from step *s* forward. This row is required for the detection of *MRF* events. For example, in step 3, both of the existing MSHRs are reserved by the previous accesses (references to blocks *a* and *b*) and the new reference to block *c* fails to reserve an MSHR. In the next step, the access retries to acquire an MSHR, and since the reserved MSHR by the access to block *a* is released at the end of step 3, the access to block *c* can proceed to reserve an MSHR. The access in step 6 requests block *c*, which is not yet transferred to the cache. Since an ongoing request to block *c* exists in an MSHR, the new access merges into the existing one.

4.2.3 Cache Associativity Modeling. The basic RDA method can be used as a basis to analyze fully associative LRU caches. Below, the RDA method is extended to model the effects of cache associativity. In a set-associative cache, the reuse distances should be calculated per cache set. Moreover, the limited number of pending misses per cache set should be modeled. Each missed access reserves a block in its respective cache set. When all the blocks in the set are reserved by the ongoing misses, no further missed access to the cache set can be handled, thereby leading to a *BRF*. A *BRF* event stalls the pipeline that issued the access. Table 3 shows the detailed RDA applied to a sample access sequence when cache associativity is modeled. Further, the assumed cache organization is depicted in Figure 4. Here, MSHRs are assumed to be infinite. The third row shows the set to which the references are mapped. The cache has two sets, denoted by S0 and S1. In the example, *a*, *c* and *e* addresses are assumed to be mapped to the first cache set (S0) and the other addresses to the second set (S1). The fourth row (#res. Blocks) represents the number of reserved blocks in the respective cache set (i.e., S0 and S1). An upward/downward arrow in this row represents an increase/decrease in the number of reserved cache blocks. This row is required to determine *BRF* events. The sixth row indicates the outcome of the access analysis and the last row illustrates the arrival of the requested blocks.

Step 3 illustrates the situation in which a *BRF* event occurs. In this step, the requested data, *e*, is not present in the cache. Further, since the reference is mapped to S0 and both of the blocks in S0 have been already reserved (number two in the fourth row), no available block exists in the cache set, thereby leading to the occurrence of a *BRF*, as shown in the sixth row.

Fig. 5. The cache event flow diagram in Reference [27] (upper) and the present study (lower).

It should be noted that in fully associative caches, despite the possibility of the occurrence of *BRF* events, *MRF* events are dominant when MSHRs are modeled.

4.2.4 Putting Together. The lower diagram shown in Figure 5 represents the possible outcome (state) of analyzing an access when MSHRs and cache associativity are modeled. The upper diagram in Figure 5 shows the events modeled in Reference [27], in which the *BRFs* and MSHR merge limits are not modeled. The events are explained as follows (in accordance with the labels in Figure 5):

- (1) *hit*: The memory reference hits the cache.
- (2) The memory reference misses the cache, and the MSHRs are then checked to define whether misses to the same address are ongoing;
 - (a) One or more ongoing missed references to the same address exist in MSHRs (MSHR hit). In that case, the number of merged references in all such MSHRs is checked to determine whether or not a new merge is possible in one of them;
 - (i) *merge_fail*: The maximum number of merges in the MSHR entries is reached, meaning that the memory reference fails to merge into a pending reference. In this case, the analysis continues.
 - (ii) *merge*: If among the MSHRs in which the address is contained, then at least one MSHR exists and the maximum number of merges is not reached, the memory reference merges into that MSHR.
 - (b) No existing MSHR contains the same missed address (or the merge limit is reached in all the MSHRs that contain the address). Thus, the number of reserved cache blocks is checked in the respective cache set to define whether a free block exists or not,
 - (i) A free cache block exists. MSHRs are then checked to see whether or not a free MSHR exists,
 - (A) *res*: A free MSHR entry is available, so the reference reserves one. In this case, the proper latency is calculated for the missed reference, based on which, the cache state will be updated (see Section 4.4).
 - (B) *MRF*: No free MSHR entry is available. Therefore, the reference fails to reserve an MSHR.
 - (ii) *BRF*: No free cache block is available in the set.

In the case of an *MRF* or a *BRF*, the failed reference stalls and retries in the next cycles.

It should be mentioned that in the present study, a write-evict policy is assumed in the case of an L1 write hit. In this case, the data is evicted from the L1 cache.

4.3 Detailed RDA Calculation Overview

As mentioned earlier, the L1 caches in GPUs are implemented as private caches serving SMs internally, while the L2 cache is shared among all the SMs. To apply RDA to GPU cache reference sequences, a private reuse stack for each L1 cache and a shared stack for the L2 cache should be constructed. In the detailed RDA algorithm, all the stacks should concurrently be constructed and used in the RDA algorithm. Although one approach is to separately calculate the reuse profiles of each SM, this method fails to effectively model the L1-L2 interactions. Further, cache coherency protocols and atomic instructions cannot be modeled. In the present study, the reuse stacks are calculated concurrently to provide the ability of modeling the effects of the L2 cache on L1 miss latency.

4.4 Latency Modeling

When a reference misses a cache and reserves an MSHR, the state of the cache cannot be immediately updated due to the backing store access latency—the L2 cache and the main memory. The basic RDA algorithm does not originally model latency. However, to model the cache performance, the backing store latency should be modeled. The cache state is updated based on the latency of misses. Thus, it can alter the acquired RD profiles. In the present article, a proper latency model was developed to accurately model the effects of latency in accessing the L2 and main memory. In each step, the latency of accessing the next memory level can be estimated based on the available dynamic statistics of the memory system. Equation (1) can be used to estimate the latency of accessing the L2 cache:

$$l_i = K_1 + K_2 \times \frac{f(i)}{L2_parallelism}, \quad (1)$$

where l_i is the estimated latency of accessing the L2 at step i , and K_1 is a constant value that represents the minimum achievable latency. Further, K_2 is a machine-dependent constant and $f(i)$ models the congestion imposed by the kernel on the memory system at step i . $L2_parallelism$ is the L2 cache parallelism (e.g., the L2 partition count). In the present study, $f(i) = \#ongoing_misses(i)$ is assumed, where $\#ongoing_misses(i)$ represents the total number of ongoing misses in all MSHRs (of all SMs), at step i . It should be noted that the latency can reorder the references. When the latency at step i , i.e., l_i , is estimated, Equation (2) is used to assign a proper latency to reference j :

$$l_j = l_i + m_{L2}(j) \times (l_{L2_miss}). \quad (2)$$

Here, l_j is the assigned latency value to memory reference j , where memory reference j is a missed reference, which is serviced in step i . Furthermore, $m_{L2}(j)$ equals one if reference j misses the L2 cache and zero if it hits. In addition, l_{L2_miss} represents the latency value of accessing the main memory from the L2 cache. In fact, when a reference misses at the L1 and reserves an MSHR, its assigned latency is l_i . Then, l_i steps later, it is directed for the analysis at the L2 to define its outcome (hit, miss, or stall). In the case of hit ($m_{L2}(j)=0$), the respective L1's state is updated. If the outcome is a miss, then updating the L1's state is postponed for another l_{L2_miss} steps. The value of l_{L2_miss} is defined using an equation similar to Equation (1), wherein the main memory serves as the backing store.

4.5 Implementing RDA Methods

A private stack should be constructed for analyzing each private cache memory (L1). Further, a shared stack is required for the L2 cache analysis in DRDA. Each new address reference to a cache

Fig. 6. An example of the implemented AVL trees.

is searched in the cache's stack to calculate its RD value (i.e., the distance from the current location of the address—if any—to the top of the stack). If the address is not resident, then the RD value of the reference equals ∞ . The search is the most time-consuming operation in the algorithm. The stack is updated through putting the new address at the top of the stack. A naive implementation requires $O(NT)$ time to process a trace of T references containing N unique data. Using tree-based implementations can accelerate the reuse distance calculations.

In the present work, AVL trees were employed to implement the proposed RDA algorithms. The implemented AVL trees are organized based on the accessed cycles. Each node in the tree holds for each cycle the number of addresses referenced in that cycle. Additionally, a weight is assigned to store the size of the node's descendants. The weight is used to calculate the RD values using a single search in $O(\log(N))$ time. Overall, the cost of processing a trace of T references is $O(T\log(N))$. In addition, the cycle in which the accessed address was referenced for the last time should also be known. In previous works [11, 27], a hash table of $O(N)$ size and $O(1)$ time have been used to hold the last accessed cycle of each address.

4.5.1 Tree Implementation of HLRDA. The L1 traces are analyzed one by one. We used a hash table to find the last cycle in $O(1)$ and an AVL tree for each trace to calculate RD values. The tree is organized with cycles. Further, each node maintains a weight parameter showing the size of its sub-trees, which is used for calculating RD values.

4.5.2 Tree Implementation of DRDA. If hash tables are used for storing the last cycles of each data, then $P + 1$ hash tables are required for P L1 caches and one L2 cache. However, this approach is not efficient in terms of the required space. An alternative is trading time for space. In our implementation, another set of AVL trees—with addresses as the search key—was implemented for the L1 caches to store the last accessed cycles (named last cycle trees).

Figure 6 illustrates the organizations of the trees for the shown sample reference sequence. The tree on the left of the figure shows the last cycle tree and the tree on the right shows the RD calculation tree. Note that the address count in each node of the RD calculation tree is required, since more than one missed reference may arrive at the same cycle (especially if the cache is parallel). To calculate the RD value of a given reference to an address, first, the last cycle is retrieved from the last cycle tree. Then, RD is calculated through traversing the RD calculation tree.

Since the detailed RDA is hardware-specific, only calculating the reuse distance values that are less than the cache associativity is of interest, and all references with a reuse distance of K

and higher (except for the accesses with infinite reuse distance) can be accumulated to show the capacity/conflicting misses. Consequently, the length of the RD calculation trees can be limited so that the tree only contains K last accessed data. This is done through trimming the trees repeatedly to prevent their constant growth. Since trimming the trees is time-consuming, we set a threshold for the maximum tree weight. When the weight of a tree reaches the defined threshold, the tree is trimmed to weight K . In our experiments, the threshold was set to $2K$. By doing so, the search time is significantly reduced. An alternative to the trimmed trees is using the linked lists with the maximum length equal to K . However, this method is not efficient for caches with high degrees of associativity.

Modeling Cache-level Parallelism in DRDA. The GPUs that run many concurrent threads can benefit from cache parallelism, e.g., providing multiple access ports or implementing caches as multiple banks. Both of the mentioned parallelism types can alter the achieved RD profile. Hence, it is imperative for modeling. In the present study, the number of access ports is configurable. Let p be the number of ports for each cache, then, p references can be analyzed in each cycle. Consequently, the address count in the respective tree node is greater than one. Modeling multiple cache partitions/banks is straightforward, and they are modeled through defining an address to partition/bank mapping scheme (e.g., address interleaving).

5 STATISTICAL SAMPLING METHOD

Statistical sampling is a popular method toward mitigating the calculation complexity of RDA [15]. Based on the statistical sampling method, instead of applying the analysis to all input data, a minimal yet representative subset of the input is selected to be analyzed. Thus, fair speedups in the simulation time can be achieved through applying a sampling method, but at the cost of error in the results. However, the imposed errors are typically marginal.

StaStack [12] is a statistical method that transforms sparse samples of reuse time distances into miss-rate breakdown. Sparse samples can be recorded during the application's runtime, and then a statistical model is used to transform the samples to a reuse profile. As StatStack ignores the detail organization of cache memories, it can achieve significant speedups in comparison to the full RDA analysis, but at the cost of accuracy. We applied StatStack to accelerate reuse profile calculations. First, the hierarchical sampling method [12] is employed in the trace generation step (see Figure 3) to collect the sparse samples. The hierarchical sampling is separately applied to every reference sequence. Then, the StatStack method is applied to estimate the miss rates in L1 caches. The cold misses are calculated as explained in Reference [12].

It should be noted that our proposed framework transforms the GPU's thread-level parallelism to serialized cache reference sequences. The hierarchical sampling can be applied to each trace during the trace generation. The evaluation results obtained by applying the StatStack method are presented in Section 6.4.

6 EXPERIMENTAL METHODOLOGY

In this section, we evaluate DRDA and HLRDA in terms of accuracy and performance. Real GPU executions are used for testing the accuracy of the proposed methods and also they are compared with a similar model that is a detailed GPU cache model based on RDA, proposed in Reference [27] (here, we call it DCMRD). Note that no sampling technique is applied in this step. In the next step, we use GPGPU-Sim to validate the detailed results provided by DRDA. Then, we use DRDA to explore different architectural parameters in GPUs. Finally, we report the results of replacing DRDA with StatStack in the framework, in terms of speed and accuracy. The Polybench/GPU [30] and Rodinia [6] benchmark suites (see Table 4) were used in our evaluations. The kernels

Table 4. Used Workload in the Evaluations and Their RDA (DRDA and DCMRD [27]) and GPGPU-Sim Slowdowns with Respect to the Executions Performed on a GTX 970 GPU

Application (size)	Kernels' abbreviation	Slowdowns ($\times 1000$)		
		DRDA	DCMRD	GPGPU-Sim
2Dconvolution [†] (4096)	2DC	3462.8	-	3084
Matrix Mult. [†] (384)	MM	60.3	211.7	1068
3DConvolution [†] (256)	3DC	49.7	555.3	3409
ATA [†] (4096)	ATX1, ATX2	37	417.9	6546
BICG [†] (4096)	BI1, BI2	30.4	241.5	2663
Correlation [†] (256)	COR	2.5	-	190
Covariance [†] (256)	COV	2.4	-	219
FDTD [†] (2048)	FD1, FD2, FD3	348.5	262	3887
GESUMMV [†] (4096)	GES	43.6	634.9	5043
Gram-Schmidt [†] (128)	GSC	0.6	-	91
MVT [†] (4096)	MVT1, MVT2	28.7	235.3	4810
SYR2K [†] (256)	SYR2	62.2	517.1	9900
SYRK [†] (256)	SYR	36.6	273.9	4085
Backprop. [‡] (262144)	BP1, BP2	115.5	62	800
CFD [‡] (0.2M)	CFD	50.5	349.3	1626
Hotspot [‡] (1024)	HS	13.6	37.8	817
NW [‡] (4096)	NW1, NW2	7.1	59.4	819
SRAD_V2 [‡] (2048)	SRD1, SRD2	69.7	43.6	639
Average		245.7	278.7	2761

[†]From Polybench/GPU [30]; [‡]From Rodinia V3.1 [6]

of the former benchmark suite heavily rely on the hardware managed cache memories. Rodinia kernels, however, are optimized for shared and texture memories. The applied computer in our experiments had a Core i5 CPU, 8GB of RAM, with Ubuntu 14.04 OS and CUDA SDK 7.0. In the following, before presenting the evaluation results, the performance metrics used for reporting the results are introduced.

6.1 Miss-rate Breakdown

The used performance metrics in the present study are as follows:

- (1) *rd0*: represents the fraction of memory references with a reuse distance value in $[0 - K)$, where K denotes the cache associativity. Such references hit in the cache.
- (2) *rd1*: represents the fraction of references with a reuse distance in $[K - \infty)$ (capacity and conflicting miss rates).
- (3) *rd2*: represents the fraction of references with the reuse distance equal to ∞ (cold miss rate).
- (4) *BRF*: represents the number of cache block reservation fails.
- (5) *MRF*: represents the number of MSHR reservation fails.
- (6) *merge*: represents the number of merges occurred within MSHRs.

The metrics *rd0*, *rd1*, and *rd2* are suggested by Wang and Xiao [35]. This terminology is valuable, since it provides a good insight into the exploited locality in a kernel executed on a GPU. For example, kernels with high portions of *rd1* misses suffer from thrashing, since most of the cached data are replaced at some point. Thus, to design an efficient cache organization, minimizing *rd1*

Fig. 7. Miss rates in L1 caches of GTX 970 GPU versus miss-rate breakdowns obtained by (a) DRDA, (b) HLRDA, and (c) DCMRD.

Fig. 8. Profiled and estimated miss rates (using DRDA) in the L2 cache of GT 740M GPU.

and *rd2* misses can be dealt with separately, e.g., dynamic bypassing or advanced replacement techniques to alleviate *rd1* misses and pre-fetching to ameliorate *rd2* misses [35].

6.2 Testing the Model's Correlation with the GPUs

HLRDA and DRDA were applied to analyze the performance of cache memories in a GTX 970 and a GT 740M GPU. The Maxwell GTX 970 GPU has 13 SMs, two 24KB L1 cache slices per SM, 1.75 MB of L2 cache (seven banks, 256KB each); and the Kepler GT 740M GPU has two SMs and 512KB of L2 cache (L1 caches are disabled). We used NVIDIA's NVPROF to obtain the profiling results related to the cache memories.

The input parameters to the framework were set according to the available GPUs specifications provided by NVIDIA and some low-level information extracted by researchers [25]. Figure 7 represents the L1 miss rates calculated using DRDA, HLRDA, and DCMRD [27] when simulating the GTX 970 GPU. The bars on the left of the figure show the miss rates of real executions obtained by NVPROF. Further, Figure 8 depicts the L2 miss rates in GT 740M calculated by DRDA. In addition, Figure 9 shows the portions of addresses merged in L1 and L2 MSHRs of the GTX 970 GPU calculated by DRDA. In most of the benchmarks, L1 merged accesses were higher than that of the L2 (average of 8.9% L1 merges versus 3.1% L2 merges).

Fig. 9. The percentage of merged addresses at the L1 and L2 caches of GTX 970 GPU.

The simulation results indicate that DRDA has a fair accuracy in estimating cache performance of different GPUs: 3.86% miss-rate difference in L1 caches of the Maxwell GPU and 6.4% in L2 miss rates of the Kepler GPU. In terms of performance, on average, DRDA was 246,000 $\times$ slower than the Maxwell GPU and it was 11 $\times$ faster than GPGPU-Sim (see Table 4).

Comparing DRDA with HLRDA. We employed DRDA and HLRDA to investigate the extent to which modeling the detailed hardware parameters affect the accuracy of an RDA method. In this experiment, the same traces were analyzed by DRDA and HLRDA. Note that HLRDA can only be used to analyze the fully associative cache memories. For DRDA, the applied cache configuration was set based on the GTX 970 GPU specifications. For HLRDA, the calculated RD profiles were used to calculate the miss rates in 24KB fully associative caches. In terms of accuracy, while DRDA had an average miss-rate error of 3.86%, HLRDA showed an average error of 16.9% (with regard to the profiled performance on a GTX 970 GPU). Moreover, to examine the effects of modeling MSHR and block reservation fails on the accuracy of DRDA, we ran it with infinite MSHRs, where merging was disabled, and no block reservation fail was modeled. The simulated caches in this step were set-associative and their backing-store latencies were modeled. In this case, the average error in L1 miss rates was increased from 3.86% to 5.3%. In terms of execution efficiency, on average, DRDA was 246,000 $\times$, and HLRDA was 257,000 $\times$ slower than the real executions on GTX 970 GPU. It should be noted that the overhead of reference sequence generation step is common in DRDA and HLRDA. When the trace generation overhead was excluded, DRDA was 2.1 $\times$ faster than HLRDA, since DRDA limits the weight of the RD calculation trees.

Comparing DRDA with DCMRD. As shown in Table 4, in the majority of the benchmarks, DRDA was performed faster than DCMRD. DCMRD failed to simulate 2DC (which has millions of threads), COR, COV and GSC (which consist of numerous per-thread references). The mentioned benchmarks were simulated for a limited number of threads and memory references. Hence, they were excluded from the slowdown calculations. In terms of accuracy, while the average error of DRDA was 3.86%, DCMRD showed an average error of 19.5%. One potential source of error in DCMRD is that it does not model the write-eviction, which can result in performance deviations when simulating write-intensive kernels. Further, in DCMRD, no limitation is put on the number of allowed merges per MSHR and also block reservation fails are not modeled. It should be noted that DRDA and DCMRD are both hardware-specific methods.

6.3 Architectural Space Exploration

One usage of the proposed framework is architectural space exploration. Various architectural parameters should be studied when designing cache memories. The proposed framework can be used to investigate the effects of different hardware parameters on cache performance. In this section, we first validated the results provided by DRDA with GPGPU-Sim.

6.3.1 Validating DRDA Against GPGPU-Sim. BRF, merge, and MRF events cannot be acquired using NVPROF. To validate DRDA in generating these events, we use GPGPU-Sim simulator. The

(a) Miss rates and merge percentages calculated by GPGPU-Sim (left) and DRDA (right)

(b) Reservation fails calculated by GPGPU-Sim and DRDA

Fig. 10. Comparison of GPGPU-Sim and DRDA.

applied configuration in the simulations of this step is as follows: 8 SMs, two schedulers per SM, 32KB L1 (four-way set-associative) with 128B blocks, 768KB L2 (8-way set-associative), 48/16 warps/blocks per SM, 32 MSHRs, and WGTO warp scheduling policy. We simulated all of the kernels using DRDA and GPGPU-Sim. Figure 10(a) illustrates the resultant miss rates and merged addresses (pending hits in GPGPU-Sim). Further, Figure 10(b) illustrates reservation fails, divided by the total number of accesses generated by the kernel. As the results indicate, DRDA and GPGPU-Sim generate similar miss rates. The calculated reservation fails are also proportional. However, in some kernels, there is a significant difference in the calculated reservation fails. Correlating reservation fails in DRDA with that of the GPGPU-Sim is hard, chiefly because DRDA relies on memory trace analysis while GPGPU-Sim analyzes a whole kernel. We performed another experiment wherein the L1 cache size was set to 16KB. Only MM, ATX1, GES, MVT1, MVT2, NW1, and NW2 kernels were included. With respect to 32KB L1 size, we observed an average of 54% increase in the reservation fails calculated by DRDA and an average of 29% increase in the reservation fails calculated by GPGPU-Sim.

6.3.2 Using DRDA for Architectural Exploration. We used DRDA to examine the effects of different hardware parameters, including different numbers of L1 MSHRs, multiple GPU schedulers/L1 ports, and different thread scheduling policies on cache performance. The applied configuration is the same introduced in the earlier paragraph, unless otherwise stated. The evaluation results are presented below.

MSHRs. Figure 11 shows the percentage of merged addresses at L1 caches along with *MRF* and *BRF* events occurred when changing the number of L1 MSHRs from four to 64. The size of the L1 caches was set to 16KB. Only the results of several representative kernels are included in Figure 11. As it can be observed, the kernels represent different behaviors. Additionally, when the MSHRs are few (e.g., eight), *MRF* events are dominant. Moreover, in most of the kernels, for 64 MSHRs, *BRF* events are significantly higher than *MRF* events. In this experiment, the observed changes in the miss rates were within 5%.

Fig. 11. Cache performance for different numbers of L1 MSHRs.

Fig. 12. merge, BRF, and MRF events plot for different numbers of schedulers/L1 access ports (S/P).

Schedulers/L1 Access Ports (S/P). Four configurations including 1/1, 2/2, 4/2, and 4/4 schedulers/ports were tested on a selection of kernels. The results are illustrated in Figure 12. Generally, by increasing the number of schedulers/access ports, *merge* counts are increased. The same is true in most of the kernels for *MRF* and *BRF* events. As it can be observed, while *BRF* events are dominant in some kernels (e.g., MM), the reverse is true in some other kernels. Moreover, in the case of four ports, the results indicate a significant increase in the number of *MRF* stalls. This can be alleviated through increasing the cache capacity, or employing the cache bypassing techniques.

Warp Scheduling Policies (see Section 3.2). On average, with respect to WGTO, the miss rate was increased by 2.4% when WILRR was applied. In addition, *BRFs* were decreased by 1.5% and *MRFs* were increased by 2.2%, on average. When WRR applied, the miss rate was reduced by 0.6%, and both *BRFs* and *MRFs* were decreased by averages of 0.94% and 3.7%, respectively.

CUDA Block to SMs Mapping Schemes. When CUDA blocks were randomly distributed among the SMs (BRND policy), on average, *rd1* was increased by 1.3% and *rd2* was decreased by 0.8%, respectively. Similarly, when BPRT was used instead of BRR, on average, *rd1* was increased by 1.5% and *rd2* was decreased by 0.9%, respectively.

Fig. 13. StatStack sampling errors in L1 caches (with a sampling rate of 2.10^{-3}) with respect to DRDA.

6.4 Statistical Sampling Results

Instead of analyzing a full trace by means of RDA, StatStack can be applied to estimate the reuse profiles. In this case, the reference sequence generation in the framework is augmented to employ the hierarchical sampling whereby capturing sparse reuse samples. Then, the StatStack model (see Section 5) is utilized to estimate the reuse profiles. We tested two sampling rates including 2.10^{-3} and 2.10^{-4} . In this step, both DRDA and the StatStack methods were employed to model the fully associative L1 caches of 16 KB size. Other simulation parameters are the same ones that were introduced in Section 6.3.1. When the overhead of trace generation is excluded, StatStack performs 397X (2.10^{-3} sampling rate) and 1,497X (2.10^{-4} sampling rate) faster than DRDA. Figure 13 shows the StatStack's errors with respect to the DRDA method. The average errors in *rd1* were 11.4% and 11.5%, and in *rd2* were 15.2% and 17.6%, for the sampling rates of 2.10^{-3} and 2.10^{-4} , respectively.

6.5 Discussion

The results obtained using DRDA and HLRDA indicate that ignoring the detailed cache-related parameters can cause performance skews. The DRDA method takes important detailed parameters into account thereby accurately analyzing the GPU cache hierarchy. Further, DRDA enhances the field through improving the state-of-the-art algorithm in terms of accuracy and speed. However, since it models the architectural parameters, the resultant RD profiles are hardware-dependent. Therefore, the generated reuse profiles are only accurate for the simulated cache. However, the reuse profiles acquired by HLRDA are general and can be used for estimating the miss rates in fully associative caches of all sizes. Our evaluation results reveal that ignoring the detailed parameters to enhance generality of the results can significantly deviate the estimated performance in some kernels. A benefit of a high-level RDA algorithm is that it can be substituted with statistical sampling. Applying sampling methods to DRDA is more challenging and deserves to be studied in the future.

7 RELATED WORK

RDA for Multicore CPUs. Due to the prevalence of multicore CPUs in modern computer systems, adapting RDA for multicore CPUs has been studied in many works. In multicore CPUs, the memory interleaving, which is enforced by concurrent threads, alters the acquired RD profiles. Further, the inter-thread data sharing can change the RD profile. Ding and Chimbili [10] modeled the memory interleaving and inter-thread data sharing to acquire RD profiles in multicore CPUs with shared caches. Similarly, Jiang et al. [19] introduced the concurrent reuse distance (CRD) profile. They used probabilistic models for the estimation of CRD profiles from thread's individual memory traces. Schuff et al. [32] further extended RDA for both private and shared caches. They also modeled write-invalidation in private caches and data-sharing in shared caches located in different cores. By modeling the hardware-related parameters, such as thread interleaving, the acquired RD

profiles are hardware-dependent (e.g., RD profiles depend on the relative speeds of cores). Wu and Yeung [36] argue that CRDs of parallel loops, in which different threads behave similarly in terms of memory referencing, can be considered as virtually independent profiles. The authors used CRD profiles obtained for a given number of cores on large-scale chip multiprocessors (LCMPs) to predict the reuse profiles when scaling the core count. Later, Wu and Yeung included private reuse profiles (PRD) in their model to explore the cache architectural space in multicore CPUs [37, 39]. They developed analytical models that used RD profiles to estimate the average memory access time. Recently, Badamo et al. [3] proposed analytical performance models to estimate performance and power consumption in LCMPs from RD profiles.

To mitigate the computation complexity of RDA, Wu & Yeung used some prediction models to avoid RDA calculation for all possible core counts in LCMPs [38]. Based on their method, only the RD profiles of several hardware configurations are required to predict the RD profiles of all desired hardware configurations. Further, the statistical sampling [31, 41] and approximate calculation [11] methods and parallelizing techniques [7] were also utilized for RDA acceleration.

GPU Performance Modeling. Hong and Kim introduced an analytical performance model (CWP/MWP) [16], but in their model, the cache memories were disregarded. Later, Sim and Kim [33] extended the CWP/MWP model to include the effects of cache memories, but their model required the kernel's hit rate as an input. In another study conducted by Bagsorkhi et al. [4], a hierarchical memory model was proposed based on the statistical sampling and trace file analysis to predict the performance of the GPU memory hierarchy. The authors used the Monte Carlo simulation method to obtain appropriate memory access sequences. Huang et al. introduced GPUMech [17], in which the interval analysis technique was used. GPUMech models GPU multithreading, MSHRs, and DRAM bandwidths. The round-robin (RR) and greedy-then-oldest (GTO) policies were employed to determine the order of memory accesses in a GPU trace. Dao et al. [9] show that linear analytical models fail to capture the nonlinear effects of GPU memory systems and presented a machine learning-based model for GPUs to accurately estimate the performance of OpenCL kernels. In Reference [21], a model is proposed to approximate the data locality in CUDA kernels with regular access patterns. In most of the GPU performance models, no detailed model is developed for cache memories and in many of them the cache behavior, in terms of hit/miss, is an input to the model.

RDA for GPUs. Tang et al. adapted RDA for GPUs and used it for L1 cache analysis [34]. The authors do not provide any details about how they modeled the GPU's physical limitations. Recently, Wang and Xiao applied RDA to analyze the access patterns of GPU applications based on the achieved RD profiles [35]. Their model relied on GPGPU-Sim to obtain memory access information. Thus, a considerable amount of time had to be spent for memory trace extraction and then for RD profile calculation. In Reference [27], the basic RDA algorithm was extended for GPUs to analyze L1 cache memories. In addition, trace files were acquired using Ocelot [13], and cache reference sequences were obtained based on the RR scheduling policy. However, the authors excluded the cache store references from the analysis and the cache-level parallelism was also ignored in their model. Additionally, to reduce RDA calculation time, the authors calculated the RD profile of one SM and generalized it to the whole GPU. This method can produce misleading results when the memory references in different SMs exhibit diverse access patterns. The authors also modeled MSHRs, but ignored the cache *BRFs*. Moreover, they introduced miss latency; however, the authors employed a simple probabilistic latency model. In the present study, we introduced a more detailed cache analysis method, which models MSHRs and cache block reservation fails. Further, the L1-L2 interaction was modeled to more accurately estimate the backing store latency. In addition, we modeled the cache stores and also investigated the effects of different cache parameters on

the obtained RD profiles. One major limitation of our approach (and similar detailed RDA models) is that the acquired RD profiles cannot be applied to GPUs with different hardware resources, because GPU-specific parameters were modeled.

8 CONCLUSION

In this article, we proposed a framework to provide RD profiles for GPUs. The framework generates cache traces and applies a trace-driven analysis, e.g., an RDA method, to obtain cache performance metrics. We employed two RDA methods called DRDA and HLRDA to study the extent to which modeling the detailed hardware parameters affects the accuracy of miss-rate estimation. The results of evaluations indicated that the average miss-rate error estimated using DRDA was less than 4%, on average. In addition to miss-rate breakdowns, DRDA reports several important performance-related metrics including MSHRs and cache block reservation fails. The main limitation of DRDA is that its generated reuse profiles are hardware-specific. By ignoring the hardware-related parameters, one simulation run of HLRDA can be applied to calculate the miss rates of all cache sizes. The average error of miss rates generated by HLRDA was 16.9%. We used AVL trees to implement the RDA methods. For DRDA, we used the trimmed trees that had a limited depth and accelerated the RD calculations. DRDA and HLRDA were 246,000× and 257,000× slower than a Maxwell GPU. However, DRDA has higher overall performance and accuracy than that of the state-of-the-art RDA method. The evaluation results also indicate that high-level sampling methods can significantly accelerate the RDA calculations. In our experiments, the time spent to generate the ordered cache traces to model the GPU's thread-level parallelism was considerable. Therefore, accelerating the trace generation for GPUs should be addressed in future studies. Another challenge that requires further research is developing some efficient statistical sampling methods for detailed RDA.

REFERENCES

- [1] George Almási, Călin Cașcaval, and David A. Padua. 2002. Calculating stack distances efficiently. In *Proceedings of the 2002 Workshop on Memory System Performance (MSP'02)*. ACM, New York, NY, 37–43.
- [2] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 136–150.
- [3] Michael Badamo, Jeff Casarona, Minshu Zhao, and Donald Yeung. 2016. Identifying power-efficient multicore cache hierarchies via reuse distance analysis. *ACM Trans. Comput. Syst.* 34, 1 (2016), 3.
- [4] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W. Hwu. 2012. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 23–34.
- [5] Kristof Beyls and Erik H. D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*. 617–662.
- [6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, 44–54.
- [7] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng. 2012. A highly parallel reuse distance analysis algorithm on GPUs. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium*. 1080–1092.
- [8] Hongwen Dai, Chao Li, Huiyang Zhou, Saurabh Gupta, Christos Kartsaklis, and Mike Mantor. 2016. A model-driven approach to warp/thread-block level GPU cache bypassing. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*. ACM, New York, NY, Article 94, 6 pages.
- [9] Thanh Tuan Dao, Jungwon Kim, Sangmin Seo, Bernhard Egger, and Jaejin Lee. 2015. A performance model for GPUs with caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2015), 1800–1813.
- [10] Chen Ding and Trishul Chilimbi. 2009. *A Composable Model for Analyzing Locality of Multi-threaded Programs*. Technical Report. Technical Report MSR-TR-2009-107, Microsoft Research.

- [11] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, NY, 245–257.
- [12] D. Eklov and E. Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 55–65.
- [13] Naila Farooqui, Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili, and Karsten Schwan. 2011. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 9.
- [14] S. Gianelli, E. Richter, D. Jimenez, H. Valdez, T. Adegbiya, and A. Akoglu. 2017. Application-specific autonomic cache tuning for general purpose GPUs. In *Proceedings of the International Conference on Cloud and Autonomic Computing (ICCAC'17)*. 104–113.
- [15] Q. Guo, T. Chen, Y. Chen, and F. Franchetti. 2016. Accelerating architectural simulation via statistical techniques: A survey. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 35, 3 (Mar. 2016), 433–446.
- [16] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 152–163.
- [17] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S. Lee. 2014. GPUMech: GPU performance modeling technique based on interval analysis. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 268–279.
- [18] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.* 22, 1 (2011), 105–118.
- [19] Yunlian Jiang, Eddy Zhang, Kai Tian, and Xipeng Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*. Springer, 264–282.
- [20] M. Khairy, M. Zahran, and A. Wassal. 2017. SACAT: Streaming-aware conflict-avoiding thrashing-resistant GPGPU cache management scheme. *IEEE Trans. Parallel Distrib. Syst.* 28, 6 (June 2017), 1740–1753.
- [21] M. Kiani and A. Rajabzadeh. 2017. VLAG: A very fast locality approximation model for GPU kernels with regular access patterns. In *Proceedings of the 7th International Conference on Computer and Knowledge Engineering (ICCKE'17)*. 260–265.
- [22] Kyu Yeun Kim and Woongki Baek. 2016. Quantifying the performance and energy efficiency of advanced cache indexing for GPGPU computing. *Microprocess. Microsyst.* 43 (2016), 81–94.
- [23] Y. Liang, X. Xie, Y. Wang, G. Sun, and T. Wang. 2017. Optimizing cache bypassing and warp scheduling for GPUs. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* PP, 99 (2017), 1–1.
- [24] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.
- [25] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.* 28, 1 (2017), 72–86.
- [26] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. IEEE, 308–317.
- [27] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 37–48.
- [28] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. Retrieved from <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.
- [29] CUDA NVIDIA. 2011. C programming guide version 4.0. NVIDIA Corporation.
- [30] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. Retrieved from <http://www.cs.ucla.edu/pouchet/software/polybench>.
- [31] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. IEEE, 53–63.
- [32] Derek L. Schuff, Benjamin S. Parsons, and Vijay S. Pai. 2010. Multicore-aware reuse distance analysis. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. IEEE, 1–8.
- [33] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 11–22.

- [34] Tao Tang, Xuejun Yang, and Yisong Lin. 2011. Cache miss analysis for GPU programs based on stack distance profile. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS'11)*. IEEE, 623–634.
- [35] Dongwei Wang and Weijun Xiao. 2016. A reuse distance based performance analysis on GPU L1 data cache. In *Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC'16)*. IEEE, 1–8.
- [36] Meng-Ju Wu and Donald Yeung. 2011. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 264–275.
- [37] Meng-Ju Wu and Donald Yeung. 2012. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'12)*. ACM, New York, NY, 2–11.
- [38] Meng-Ju Wu and Donald Yeung. 2013. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Trans. Comput. Syst.* 31, 1 (2013), 1.
- [39] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 499–510.
- [40] Yang Zhang, Zuocheng Xing, Cang Liu, Chuan Tang, and Qinglin Wang. 2018. Locality based warp scheduling in GPGPUs. *Future Gen. Comput. Syst.* 82 (2018), 520–527.
- [41] Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*. ACM, New York, NY, 91–100.

Received January 2018; revised October 2018; accepted October 2018