Improving First Level Cache Efficiency for GPUs Using Dynamic Line Protection

Xian Zhu, Robert Wernsman, and Joseph Zambreno
Iowa State University
Ames, Iowa, USA
{xian, wernsman, zambreno}@iastate.edu

ABSTRACT

A modern Graphics Processing Unit (GPU) utilizes L1 Data (L1D) caches to reduce memory bandwidth requirements and latencies. However, the L1D cache can easily be overwhelmed by many memory requests from GPU function units, which can bottleneck GPU performance. It has been shown that the performance of L1D caches is greatly reduced for many GPU applications as a large amount of L1D cache lines are replaced before they are re-referenced. By examining the cache access patterns of these applications, we observe L1D caches with low associativity have difficulty capturing data locality for GPU applications with diverse reuse patterns. These patterns result in frequent line replacements and low data re-usage.

To improve the efficiency of L1D caches, we design a Dynamic Line Protection scheme (DLP) that can both preserve valuable cache lines and increase cache line utilization. DLP collects data reuse information from the L1D cache. This information is used to predict protection distances for each memory instruction at runtime, which helps maintain a balance between exploitation of data locality and over-protection of cache lines with long reuse distances. When all cache lines are protected in a set, redundant cache misses are bypassed to reduce the contention for the set. The evaluation result shows that our proposed solution improves cache hits while reducing cache traffic for cache-insufficient applications, achieving up to 137% and an average of 43% IPC improvement over the baseline.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**;

KEYWORDS

GPU, L1 data cache, cache management

ACM Reference Format:
Xian Zhu, Robert Wernsman, and Joseph Zambreno. 2018. Improving First Level Cache Efficiency for GPUs Using Dynamic Line Protection. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225104

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225104

1 INTRODUCTION

Graphics Processing Units (GPUs) are increasingly becoming the preferred choice of accelerator for various scientific and engineering applications [2–4, 19, 22], as they offer high floating-point throughput by executing many threads concurrently. Similar to traditional multi-threading systems, state-of-the-art GPUs also adopt data caches in each GPU core to reduce the massive traffic generated by parallel threads [9, 21]. However, recent studies show that some GPU applications cannot effectively utilize the on-chip L1 Data (L1D) caches due to high contention for cache lines [17, 24, 26]. This results in cache thrashing behavior, i.e., many cache lines are replaced before they can receive a significant number of hits, even if they will be reused in the future. Since GPU cores typically have a small L1D cache and many concurrent threads, contention for L1D cache entries is high and cache thrashing becomes a severe problem.

Since the cache thrashing behavior causes L1D caches to fail to effectively handle data re-usage in certain GPU applications, the miss rates of L1D caches are high in these scenarios. Consequently, additional memory pipeline stalls occur due to full reservations of miss handler registers or cache sets. Additionally, cache thrashing causes frequent evictions of L1D cache lines; this injects additional traffic into the interconnection network. Previous work has observed that this excessive eviction traffic also degrades GPU performance [13].

Several cache management methods have been proposed to mitigate cache thrashing behavior in L1D caches on a GPU platform. [13, 27] utilize the compiler to analyze the GPU application and estimate the cache lines that have less re-usage in the near future. However, the compiler-based techniques only leverage static application information for their usage estimations; they cannot predict cache behaviors that depend on application inputs. As an improvement, [17, 18, 26] introduce extra hardware to track the reuse frequencies of cache lines dynamically and bypass memory accesses for infrequently used cache lines. Instead of adjusting the replacement policy to preserve more data locality, these works focus on identifying cache lines with low re-usage and bypassing memory accesses to them to reduce cache traffic.

Based on the concepts of Reuse Distance (RD) and Reuse Distance Distribution (RDD), [7] devises a Protecting Distance-based Policy (PDP) for CPU platforms. This work utilizes the RDD of the program to predict a Protecting Distance (PD). This method extends the lifetime of all cache lines in the CPU's Last Level Cache. However, it is hard for a single PD to fully describe diverse reuse patterns within GPU applications. Hence, we propose a Dynamic Line Protection scheme (DLP) to manage the replacement and bypassing policies of L1D caches on a GPU. Instead of applying a single PD to all cache lines, DLP dynamically predicts an appropriate line lifetime

extension for each memory instruction based on both program-level and instruction-level reuse information. By using a simple bypass policy, the DLP scheme can improve the utilization of cache lines with minimal overhead. The major contributions of this paper are as follows:

- Through a detailed analysis on data reuse patterns for GPU applications, we make the following observations: Firstly, Data reuse patterns are diverse among different GPU applications. Insufficient L1D cache space causes many applications to suffer locality loss. Secondly, The data reuse patterns with distinct memory instructions in a GPU application can differ wildly. A single PD for all cache lines may not fit every reuse pattern of the application.
- We propose a Dynamic Line Protection solution to capture the various reuse patterns from different memory instructions at runtime. Our design collects and uses the RD information at both program level and instruction level, improving the utilization of data locality for each memory instruction and reducing the overhead of over-protection.
- Our evaluation results show that the DLP approach can increase IPC performance of Cache Insufficient applications by an average of 43.8% over the baseline configuration. In contrast, the Global Protection scheme only improves the performance by 34.7%. For Cache Sufficient applications, DLP can achieve 99.8% IPC performance compared to baseline architecture. The overall hardware cost of DLP is 7.5% of the baseline cache.

The remainder of the paper is organized as follows: Section 2 discusses the background of GPU memory hierarchy. Section 3 analyzes the limitation of current L1D cache implementations. Section 4 provides a detailed description of our design. Section 5 presents the experimental methodology. Section 6 studies various evaluation results of our design and compares them to other schemes. Section 7 discusses related work in the field of cache management. Finally, Section 8 concludes the paper and provides direction for future work.

2 BASELINE GPU ARCHITECTURE

In order to execute many threads in parallel, a GPU chip typically contains multiple cores, also referred to as Streaming Multiprocessors (SMs) in Nvidia terminology[1]. SMs process GPU workloads in a Single Instruction Multiple Threads (SIMT) fashion. Initially, SMs fetch and decode instructions for groups of threads, referred to as warps. Instructions for the warps are issued by warp schedulers based on a warp scheduling policy. The issued instructions are executed by the corresponding Function Units (FUs). Specifically, the Load/Store (LD/ST) unit processes all memory instructions, generating one or more memory data requests for each memory instruction. The SMs utilize multiple on-core caches to handle different types of memory accesses as shown in Figure 1. In this paper, we only focus on the L1D cache.

Figure 1 shows the handling process of a memory request in a L1D cache. When the L1D cache receives a memory request from the LD/ST unit, it first checks the cache hit status of this request.

[1]While we use Nvidia terminology in this paper, the main concepts generally apply to other GPU architectures.

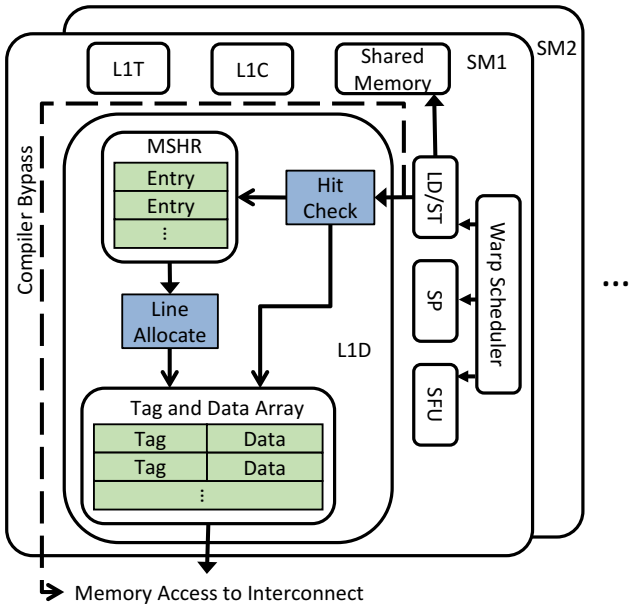


Figure 1: Baseline GPU architecture.

If the request is a hit, the L1D cache immediately responds with the cached data. Otherwise, it inspects the Miss Status Holding Registers (MSHRs) to check if the requested line has already been registered by previous requests. If so, the L1D cache appends source information, such as the request thread ID, to the corresponding MSHR entry. If the requested line does not match any previous entries, a new entry is added to MSHR. The L1D cache will then try to reserve a line in the corresponding cache set for this miss request. Due to the limited number of entries in MSHR and the cache set, the MSHR or the targeted cache set may have no reservable space left. In this case, the miss request will be blocked in the pipeline register and continue to retry in the following cycles until the block is resolved. Consequently, all future accesses to the L1D cache will be stalled.

Modern GPU architectures have started supporting L1D cache bypassing [9, 21]. Programmers can utilize compiler flags to switch to a static bypassing path as shown in Figure 1. However, these flags affect all memory requests from a GPU application. This forces the application to use only one path at runtime, wasting the bandwidth available in the alternative path.

3 MOTIVATION

3.1 Data Reuse Behavior of GPU Applications

To understand data reuse patterns of GPU applications, we make use of the concept of Reuse Distances for measuring a memory line's re-referenced interval [7]. In [7], a RD is defined as the number of other memory accesses to a cache set between two accesses to the same cache line within that set. Based on this definition, the RD is only related to the data access pattern of the program and the set mapping of the cache. In other words, associativity size will not affect RDs of a program if the number of cache sets and set mapping method do not change. Figure 2 demonstrates

Figure 2: Examples of Reuse Distances(RD). The RD of Addr 0 is 3.

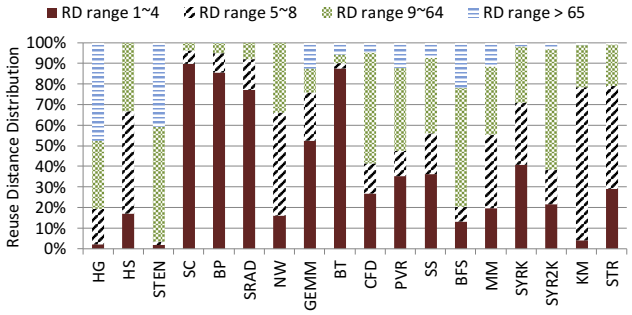


Figure 3: Reuse Distance Distribution (RDD) of different applications

an example of counting RD within a 2-way set associative cache. Under the Least Recently Used (LRU) replacement policy, there is no guarantee that future re-visits will hit the requested cache line if the RD is larger than the associativity size of the cache. In Figure 2, the second access to cache line 0 is a miss because the requested line had been replaced earlier. If most RDs within an application are larger than the associativity size of the cache, many cache lines will not be reused before they are evicted. The cache will keep swapping data in and out frequently with little data re-usage; this is known as cache thrashing behavior. Cache thrashing can lead to excessive cache misses. The penalties of cache misses in L1D of GPUs are extremely high since they block the whole memory pipeline as mentioned in Section 2.

Figure 3 illustrates the Reuse Distance Distribution for a set of benchmark applications. Here, we classify RDs into 4 different ranges; each shaded portion of the bar represents the percentage of RDs that falls into that range. Figure 3 unveils three data reuse patterns for GPU applications. First, the RDD varies significantly among different GPU applications. For example, HG and STEN have mostly long RDs, while SC, and BP have short ones. Second, many GPU applications do not fully utilize the L1D cache. The majority of RDs in applications like HG and KM are greater than the baseline associativity size of the L1D cache described in Table 1. As discussed previously, these applications may suffer from cache thrashing behaviors because the baseline cache is too small to capture data localities. Third, RDs may be distributed across all ranges. For instance, within application MM, the percentages of RDs are 19.5, 35.8, 33.2 and 11.5 for RD ranges 1~4, 5~8, 9~64, and >64 respectively.

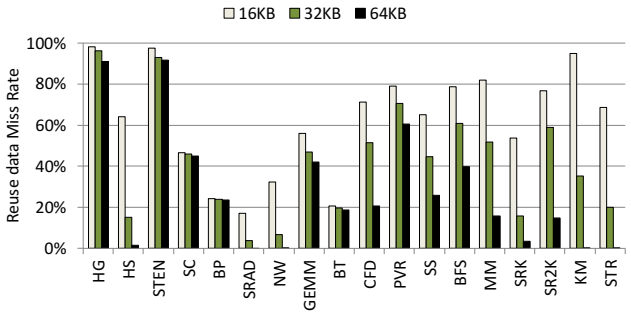


Figure 4: Cache miss rate of 16KB (4-way), 32KB (8-way) and 64KB (16-way) L1D caches.

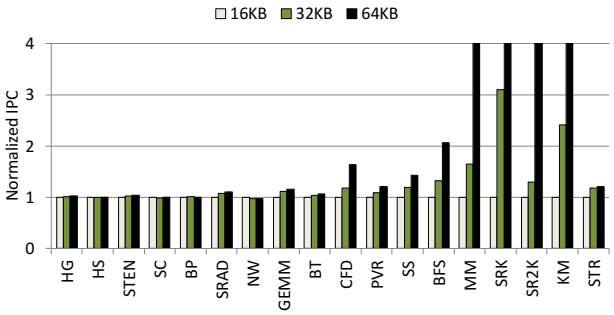


Figure 5: IPC performance of 16KB (4-way), 32KB (8-way), and 64KB (16-way) L1D caches, normalized to performance of 16KB L1D cache.

3.2 Insufficiency of the Cache Size for GPU Applications

A simple and direct method to alleviate cache thrashing behavior is increasing the associativity of the L1D cache to hold more reuse data. Figure 4 shows the miss rate of memory accesses to reuse data for a set of benchmark applications with 4-way, 8-way, and 16-way set associative caches. Note that we exclude all compulsory misses in Figure 4, as by definition these accesses will always miss regardless of the L1D cache size. We can observe that the reuse data miss rate for most applications is reduced when the associativity of the L1D cache is increased. There are some exceptions to this observation, such as HG, STEN, SC and BP; most of their RDs are grouped in either short ranges or long ranges. Figure 5 demonstrates the IPC improvements of 8-way and 16-way set associative caches normalized to a 4-way set associative cache. While some GPU applications exhibit significant IPC improvement, applications like HS and NW cannot benefit significantly from associativity increasing. The main reason for this is that memory operations of these applications only require a small portion of the overall execution time. Thus, the improvement in the L1D cache may not be reflected in the final IPC performance. We use an application's memory access ratio to estimate the importance of memory operations in this application, which is calculated using the following equation:

$$Ratio_{memory\_access} = \frac{N_{memory\_access}}{N_{insn}}$$

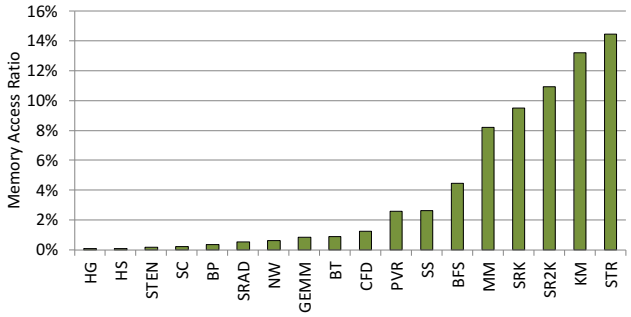


Figure 6: Memory access ratio of GPU applications.

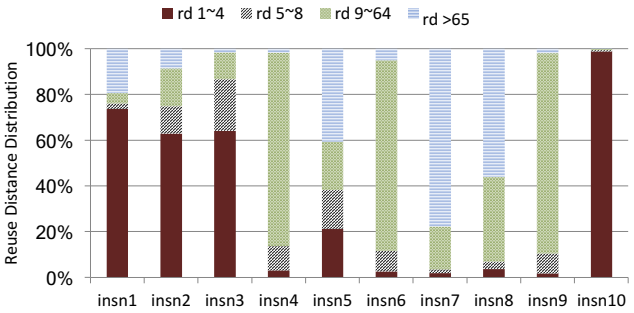


Figure 7: RDD for different memory instructions of BFS.

In the equation, N_{memory_access} is the number of memory accesses generated by the GPU application, and N_{insn} is the number of thread instructions executed by the application. Figure 6 shows sorted memory access ratios of different GPU applications. The GPU applications on the right side have higher memory access ratios, which indicates that they are more easily affected by the memory sub-system than the applications on the left side. In this paper, to better understand the impacts of the L1D cache, we empirically classify the GPU applications into two categories based on a memory access ratio threshold: Cache Sufficient (CS) applications and Cache Insufficient (CI) applications. The memory access ratio threshold is set as 1% in this paper.

Although increasing the associativity size of the L1D cache can improve the performance of CI applications, the hardware costs are high since the die size of the L1D cache is linearly correlated with its associativity. Since the cache associativity cannot be changed during runtime, it is not adaptable to all GPU applications.

3.3 Diverse Distributions of Reuse Distance Among Memory Instructions

To address the cache thrashing problem caused by a shortage of associativity, the Protection Distance (PD) method has been proposed as a solution that does not enlarge the cache size [7]. However, we observe that a single PD cannot fit diverse data reuse patterns in some GPU applications. As GPU applications execute instructions in a SIMT fashion, memory accesses from the same program counter (PC) have the same type of instruction and typically share a similar data reuse pattern. Figure 7 illustrates the RDDs for all memory instructions at different PCs in the application BFS. It is obvious that the data re-references for different memory instructions vary in RD distribution. On one hand, most RDs for instruction 2 and 3 are small; this indicates that most reuses of the cache lines brought in by these instructions can be caught by the L1D cache. Therefore, they do not need large PDs for the cache lines they brought in. On the other hand, the majority of RDs for instruction 4 and 9 fall into the range 9~64, which implies that a larger PD is needed to capture data re-usage. As a result, using multiple instruction-based PDs instead of a single PD can better accommodate diverse reuse patterns inside GPU applications.

For certain memory instructions, adjusting the PDs improve the hits for corresponding data re-references. However, if a cache line is protected for an extended period of time, it could reduce cache availability for memory accesses to other lines mapped to the same set. For instance, instruction 7 and 8 in Figure 7 bring in cache lines with large RDs. It is required to set a long protecting time to capture reuses of these cache lines; however, this prevents other cache lines from replacing them during the protected period. Based on these observations, the cache line protection scheme should keep balance between data locality exploitation and over-protection for cache lines with long reuse distances.

4 DYNAMIC LINE PROTECTION APPROACH

In this section, we propose the Dynamic Line Protection cache management system, which accommodates data reuse patterns at the instruction level. The DLP scheme can dynamically collect the hit information from the L1D cache for each memory instruction, accumulating them as global hit information. It uses this global information to enable PD adjustment, and calculates the PD values for each memory instruction based on its local information.

4.1 Architectural Modifications

DLP requires RDDs to estimate a PD adjustment for different memory instructions. However, it is hard to collect re-usage information for data with long RDs using only the baseline L1D cache, since the values of these RDs are larger than the associativity of the cache. In order to capture the information of long RDs while maintaining a low hardware cost, we modify the current Tag and Data Array (TDA) and extend the TDA with a Victim Tag Array (VTA) to hold the tags of recently evicted lines, as shown in Figure 8.

4.1.1 L1 Data Cache Modifications. In order to collect cache hit information for different load instructions, we add an instruction ID field to each TDA entry. This field records the hashed PC value of the load instruction that brought in or last hit the cache line. When a new memory request hits this cache line, the hit is attributed to the previous instruction that used this cache line. For example, consider a cache line that is originally brought into the cache by instruction 0. The cache line is then hit by instructions 1, 2, and 3 sequentially. When analyzing the cache request for instruction 3, the Instruction ID field attributes the cache hit to instruction 2 and not instruction 1 or 0. Another extra field in each TDA entry is Protected Life (PL), which is a counter for storing the extra life awarded to the cache line. Every time a memory request accesses the L1D cache, the request will read the PD value from the Protection Distance

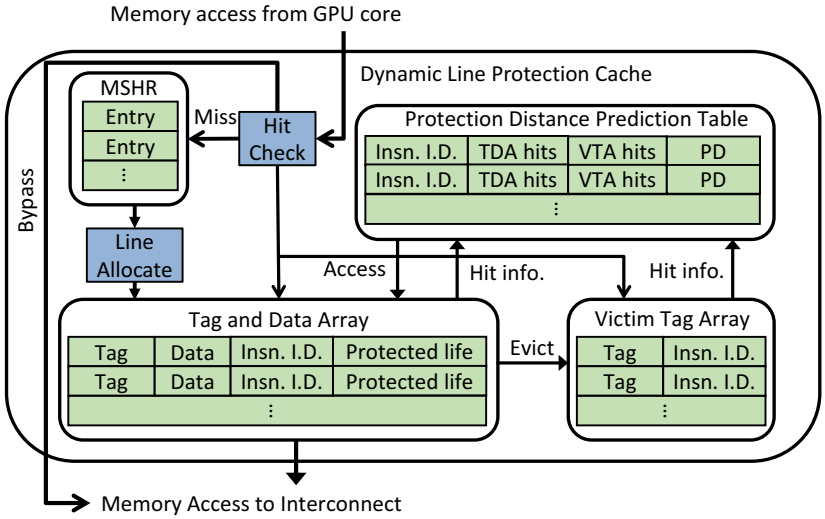


Figure 8: DLP Architecture

Prediction Table for the request's instruction and write this value to the PL field of the corresponding TDA entry.

When a set is queried, PL values of all TDA entries belonging to this set are decreased by 1. If a request needs to be allocated in a set where all TDA entries have positive PL values, the request will be bypassed since the TDA entries with PL values greater than 0 cannot be replaced. Note that these entries within a set will not be permanently locked in the TDA when they are hit infrequently, since a bypassed request also queries and consumes PL values of all entries in this set. Hence, as more requests are bypassed, some entries will eventually be released.

4.1.2 Victim Tag Array. Due to the eviction of old cache lines, it is difficult to monitor reuse patterns for the cache lines with a RD greater than the associativity size of the baseline cache. To collect reuse information for these cache lines, we add a VTA to our L1D cache design. Similar to the TDA, the VTA has multiple indexed sets where each set possesses several associative entries. The only difference between the VTA and the TDA is that entries in the VTA only contain address tags without any real data. All tags in the VTA are received from evicted cache lines in the TDA. The entries in a set are replaced in the VTA by a LRU policy. When a memory request fails to hit any lines in the TDA, it will continue searching in the VTA before being forwarded to the miss handling system. The advantage of using the VTA is that we can track the hit status for a larger data cache with only minor hardware costs. For example, to track a data cache twice the size of the baseline L1D cache, we would set the amount of VTA entries to be the same as TDA entries. However, the cost of implementing this VTA tag array is only 3% of the cost required to implement twice the number of TDA entries. Similar to the TDA, we add an instruction ID field for each victim entry in order to collect instructional hit information from the VTA.

4.1.3 Protection Distance Prediction Table. In Section 3.3, we demonstrated that the reuse pattern of cache lines is related to the source instruction. Hence, we create a Protection Distance Prediction Table (PDPT) to collect hit information from the TDA and VTA, as showed in Figure 8. This table computes a PD for each instruction based on the hit histories of the TDA and VTA. The PDPT has 128 entries; each load instruction for L1D cache has a corresponding entry inside the PDPT. Due to the limited number of entries in the PDPT, the number of load instructions for L1D cache in a GPU application cannot be larger than 128. Our experiments show that this number is much less than 128 for most GPU applications.

Each entry of the PDPT has 4 fields: The instruction ID field is used for indexing; The TDA hits field records the number of hits for each instruction in the TDA during the current sampling period; Similarly, the VTA hits field records the number of hits in the VTA during the current sampling period; The PD field calculates the life extension for the next sample, which will be described in Section 4.2. When there is a hit in the TDA or the VTA, the value of the TDA hits or VTA hits field of the corresponding entry will be increased by 1. Whenever a sample is finished, the TDA hits and VTA hits field of all entries in the PDPT is reset to 0.

4.1.4 Data Sample. We use the number of cache accesses as a counter for our sampling. The number of cache accesses needs to be large enough to accurately reflect the hit status of the TDA and VTA. As a non-trivial amount of time is required to aggregate sufficient cache accesses, we chose a sample limit of 200 based on our empirical observations. In order to save sampling time, we also reduce the relatively long sampling time for some CS applications with few load instructions by setting a maximum sample amount for the number of instructions executed. The impact of sampling time on CS applications is trivial, since their performance is rarely affected by memory accesses.

4.2 Protecting Distance Computation

The process of PD computation is demonstrated in Figure 9, which is essential to DLP. The computation starts after a sample is taken, and it compares the overall hits in the TDA and VTA. If the amount of

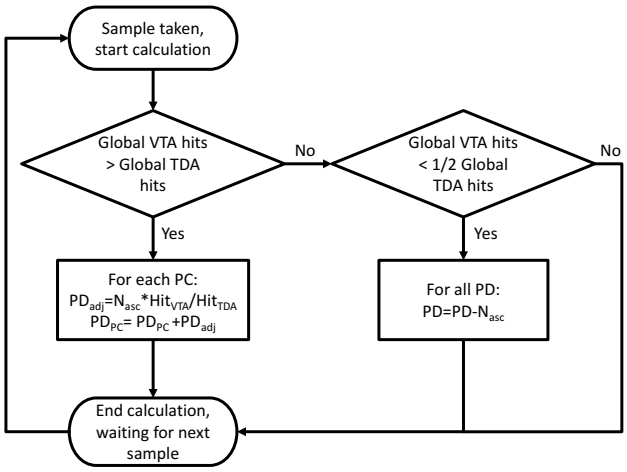


Figure 9: Flow Chart of PD Computation

overall hits in the VTA (global VTA hits) is greater than in the TDA (global TDA hits), it will follow the left path of the flowchart and increase PD, since preserving older lines longer can capture more data reuse than replacing them with new lines. If this is not the case, increasing PD is unnecessary as it may reduce the availability of TDA entries as discussed in Section 3.3. To accommodate the diverse reuse patterns among memory instructions on the PD increasing path, we calculate a PD for each memory instruction based on its own hit information in the VTA and TDA as shown in the following equation:

$$PD_{adj} = N_{asc} \times \left\lfloor \frac{Hit_{VTA}}{Hit_{TDA}} \right\rfloor$$

In the equation, N_{asc} represents the value of set associativity of the VTA[2], which is constant for a specific GPU configuration. Hit_{VTA} and Hit_{TDA} are the number of hits in the VTA and TDA that are credited to this memory instruction during the previous sample period. Thus, if a memory instruction has a larger ratio of VTA hits to TDA hits, the algorithm assigns a larger PD increment for this instruction. In our implementation, we use step comparison to avoid the high time complexity of the division operation. Specifically, we sequentially compare Hit_{VTA} with the following values generated by shifting Hit_{TDA}: four times Hit_{TDA}, two times Hit_{TDA}, Hit_{TDA}, and half of Hit_{TDA}. Then, we shift N_{asc} according to the outcome of the step comparison. Additionally, the step comparison also applies an upper limit for incrementing PD (in the previous example, 4 times N_{asc}), which can prevent over-protection. This process takes more than one cycle to complete, but is acceptable because the PD is only updated once for each sample.

If the amount of global hits in the VTA is less than half of the hits in the TDA, it is empirically estimated that the lines in the TDA have already received sufficient hits and protection for them may be unnecessary. In this case, we will decrease all PDs as the right path shows in Figure 9. Different from the PD-increasing path, we do not apply an instruction-based decision in the PD-decreasing path due to few hits in the VTA.

[2]In our configuration, the VTA associativity is set to the associativity of the cache.

Table 1: The GPU configuration used in our experiments

Number of Cores	16
Warp Size	32
Max # of warps per core	48
Warp schedulers per core	2, GTO scheduling policy
# of registers per	32768
Shared Memory	48KB
L1D cache	16KB, 32sets, 4-ways, Hash index
Core/ICNT/Memory Clock	650MHz/650MHz/924MHz
# of memory partition	12
L2 cache	768KB, 64sets, 8-ways, Linear index
DRAM Chip Model	32bits bus width/Memory Partition, 6 Banks/Memory Partition, GDDR5 timing
Memory Bandwidth	177.4 GB/s

4.3 Hardware Overhead

As previously discussed, DLP requires additional storage space in the L1D cache. For each entry in the TDA, the instruction ID field needs 7 bits and the Protected Life field needs 4 bits; thus, the extra size in the TDA for the baseline configuration is 176 bytes. Each entry in the VTA requires 32 bits for the tag and 7 bits for the instruction ID; thus, the total cache space for the VTA is 624 bytes. Each entry in the PDPT needs 7 bits for the instruction ID, 8 bits for TDA hits, 10 bits for VTA hits, and 4 bits for PD; this sums to 464 bytes of space consumed for all entries of the PDPT. In summary, the DLP scheme requires 1264 bytes of extra cache space. Considering that the baseline cache has 16896 bytes for the TDA, the overhead of our approach is only 7.48% of the baseline cache size.

5 METHODOLOGY

5.1 GPU Simulator Settings

We evaluate the performance improvement of the DLP scheme on GPGPU-Sim, a cycle-level GPU architecture simulator [1]. The simulator is configured to match the hardware settings of a Tesla M2090 [20], a Nvidia GPU with Fermi microarchitecture [9]. Table 1 provides the details of the baseline configuration used in our experiments. In addition to the baseline configuration and our approach, we also implement the Stall-Bypass and Global-Protection schemes for comparison purposes.

5.2 Experimental Benchmarks

The GPU applications used for our evaluation are chosen from Rodinia [5], CUDA Samples [23], Mars [11], Parboil [25], and Polybench [10] benchmarks listed in Table 2. As discussed in Section 3.1, we group these GPU applications as Cache Sufficient (CS) and Cache Insufficient (CI) based on their memory access ratio. CS applications are not restricted by the memory sub-system, so they can already achieve good performance with the baseline configuration. Consequently, CI applications require larger cache associativity to maintain data localities; these are good candidates for performance improvements with DLP.

5.3 Comparable Methods

Stall-Bypass: This scheme enables a bypass path when a stall is detected in the L1D cache for any reason, such as no available

Table 2: Benchmark Applications

App. Name	Abbr.	Suite	Type	Input
Histogram	HG	CUDA Samples	CS	67108864
Hotspot	HS	Rodinia	CS	512x512
3-D Stencil Operation	STEN	Parboil	CS	512x512x64
Separable Convolution	SC	Rodinia	CS	2048x512
Back Propagation	BP	Rodinia	CS	65536
Speckle Reducing Anisotropic Diffusion	SRAD	Rodinia	CS	512x512
Needleman-Wunsch	NW	Rodinia	CS	1024x1024
Matrix Multiply-add	GEMM	Polybench	CS	512X512X512
B+tree	BT	Rodinia	CS	6000x3000
Computational Fluid Dynamics	CFD	Rodinia	CI	97046
Page View Rank	PVR	Mars	CI	250000
Similarity Score	SS	Mars	CI	512x128
Breadth-First Search	BFS	Rodinia	CI	65536
Matrix Multiplication	MM	Mars	CI	256x256
Symmetric Rank-k	SRK	Polybench	CI	256x256
Symmetric Rank-2k	SR2K	Polybench	CI	256x256
K-means	KM	Rodinia	CI	204800
String Match	STR	Mars	CI	354984

MSHR entry, no reservable slot in set, or a fully occupied miss queue. The Stall-Bypass scheme eliminates delays in the L1D cache regardless of the reuse patterns of applications.

Global-Protection: This scheme uses a single PD to preserve data locality and improve re-usage of L1D cache lines, emulating the PDP method found on CPU platform [7]. It applies a PD computing algorithm similar to Figure 9; however, instead of an instruction-based PD like the left-most path in Figure 9, this scheme computes a global PD for all cache entries.

32KB L1D cache: This configuration doubles the amount of set associativity of set in the L1D cache while keeping the other parameters the same as the baseline configuration.

6 EXPERIMENTAL RESULTS

6.1 IPC Performance Improvement

Figure 10 compares the IPC performance of benchmark applications for the baseline configuration, Stall-Bypass, Global-Protection, DLP, and 32KB L1D cache. G.MEANS in Figure 10 indicates the geometric mean of IPCs of all CS applications or CI applications.

6.1.1 CS Applications. Half of the 18 applications analyzed in our experiments are CS applications. Performance improvements for CS applications are not expected for any cache management schemes; this is demonstrated in Figure 5 when significant increases to cache size does not influence application performance. Both Global-Protection and DLP can retain the performance for most applications, while the Stall-Bypass fails to do so for applications SRAD and BT. Compared to the baseline configuration, the Stall-Bypass decreases the IPC of SRAD and BT by 11% and 12% respectively. The main factor of this performance drop is the over-bypassing of the L1D cache accesses, which causes the L1D cache to miss potential data reuses. We will detail this issue in the Section 6.2. DLP performs worse than Global-Protection in the GEMM application due to over-protection from specific instructions; however, the performance loss is only 3%. As mentioned in Section 4.2, the DLP scheme also applies a global check before instruction-based

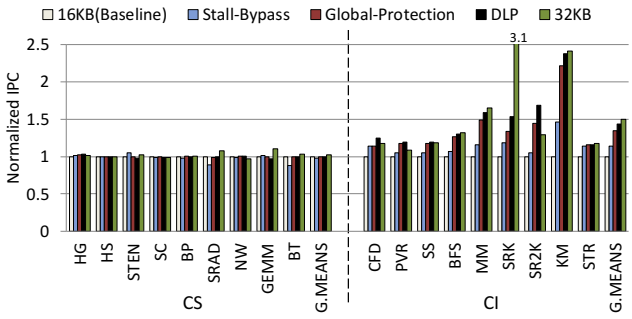


Figure 10: The IPC performance of using the baseline configuration, Stall-Bypass, Global-Protection, DLP, and a 32KB L1D cache. The performance results are normalized to corresponding IPCs achieved by using the baseline-configured L1D cache.

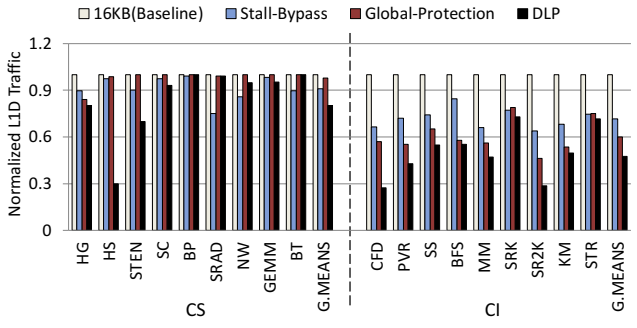
PD adjustment, so the over-protection loss is typically trivial. On average, Stall-Bypass loses 2.4% IPC performance while DLP only sacrifices 0.2%. No CS application compromises its performance by more than 3% with DLP.

6.1.2 CI Applications. For all CI applications, Stall-Bypass, Global-Protection, and DLP can improve IPC performance. The speedup of CI applications with Stall-Bypass is 1.14 on average and 1.46 at most. The improvements when using Stall-Bypass are limited , as it simply bypasses stalled memory accesses regardless of reuse patterns, miss rates, or any internal cache information. Global-Protection adjusts the global PD to accommodate the reuse patterns of GPU applications, leading to a more significant increase in IPC performance over Stall-Bypass. Furthermore, Global-Protection can achieve an average of a 34.7% performance improvement over the baseline. The DLP scheme utilizes the different reuse patterns of memory instructions in GPU applications, so it can typically achieve an even greater performance boost than Global-Protection for CI applications. Both Global-Protection and DLP can achieve better performance than the 32KB cache configuration for specific applications like CFD and SR2K; the 32KB cache configuration can typically retain data locality for 8 accesses, while Global-Protection and DLP can maintain locality much longer for prioritized cache lines. Applying DLP to a 16KB cache can achieve an average of 43.8% performance gain over the baseline configuration, which is only 6% less than the speedup achieved by doubling the set associativity.

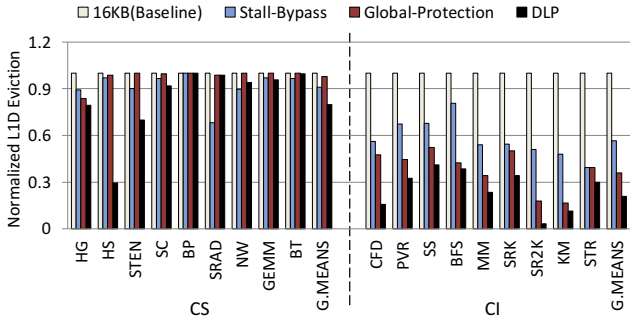
In general, the DLP scheme shares similar performance with the baseline and Global-Protection for CS applications and outperforms the baseline, Stall-Bypass, and Global-Protection schemes for CI applications.

6.2 L1D Cache Traffic

Stall-Bypass, Global-Protection, and DLP use bypasses to alleviate overwhelming accesses to the L1D cache. In this section, we will analyze the reduction of L1D cache traffic and eviction when employing these methods on baseline-configured cache. Figure 11a compares L1D cache traffic and evictions by applying the baseline

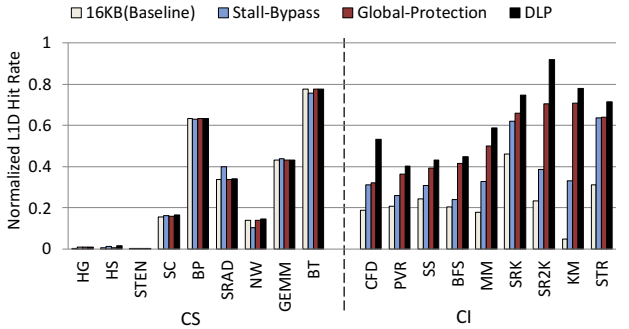


(a) Cache Traffic

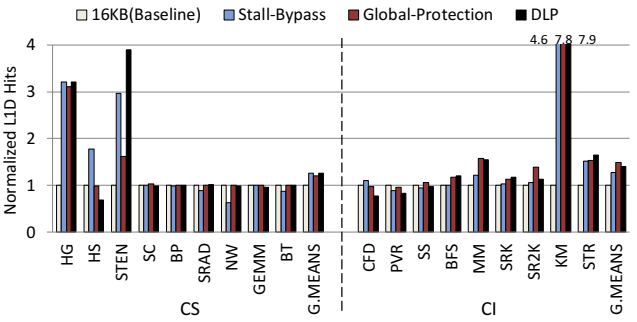


(b) Cache Eviction

Figure 11: The normalized traffic and normalized evictions in the 16KB L1D cache with baseline configuration, Stall-Bypass, Global-Protection, and DLP.



(a) Hit Rate



(b) Number of Hits

Figure 12: The hit rate and normalized number of hits in the 16KB L1D cache with the baseline configuration, Stall-Bypass, Global-Protection, and DLP.

configuration, Stall-Bypass, Global-Protection, and DLP to a 16KB L1D cache.

6.2.1 CS Applications. Both Global-Protection and DLP can maintain similar cache traffic figures for CS applications with many short RDs, i.e. SC, BP, SRAD, GEMM, and BT; both methods are able to recognize their short reuse pattern and set short PDs accordingly. As a comparison, Stall-Bypass over-bypasses for SRAD and BT, causing a performance penalty as mentioned in Section 6.1.1. Although DLP also bypasses additional cache traffic for some applications such as HG, HS and STEN, our approach still satisfies their reuse patterns without over-bypassing. As shown in Figure 3, most of these applications have a majority of long RDs. A long RD causes DLP to extend PDs for some memory instructions, resulting in many memory accesses bypassed. However, DLP's accommodation of reuse patterns for these applications yield only trivial performance increases; this due to a low memory access ratio and the large portion of compulsory misses in these applications.

As Figure 11b shows, the reductions of L1D cache eviction that Stall-Bypass, Global-Protection, and DLP provide for CS applications follow a trend similar to the reductions in cache traffic. On average, Stall-Bypass, Global-Protection, and DLP reduce 8.9%, 2.3%, and 20.0% cache eviction over the baseline configuration respectively.

6.2.2 CI Applications. CI applications typically have long RDs and a high memory access ratio, causing these applications to experience high cache contention and lengthy stall times in the L1D cache. As a result, we observe that Stall-Bypass, Global-Protection, and DLP bypass a significant amount of cache accesses in CI applications. However, the DLP scheme behaves the most aggressively when it comes to bypassing. The average L1D cache traffic with Stall-Bypass and Global-Protection schemes for CI applications is 71.6% and 59.8% of baseline cache traffic respectively, while DLP only allows 47.5% of baseline cache traffic. Although similar in pattern, the reduction of cache evictions is actually more significant than the reduction of cache traffic under the three different schemes; this is especially apparent with DLP. On average, Stall-Bypass and Global-Protection use 71.6% and 59.8% of cache traffic and generates 56.5% and 35.7% of cache evictions as compared to the baseline, while DLP consumes 47.6% of cache traffic and produces only 20.7% of cache evictions. This implies that the re-usage rate of cache lines retained by DLP is higher than that of Stall-Bypass and Global-Protection; fewer L1D cache evictions can reduce interconnect traffic, which will be detailed in Section 6.4.

6.3 L1D Cache Hits

Figure 12a demonstrates the overall hit rates of the baseline configuration, Stall-Bypass, Global-Protection, and DLP. To illustrate

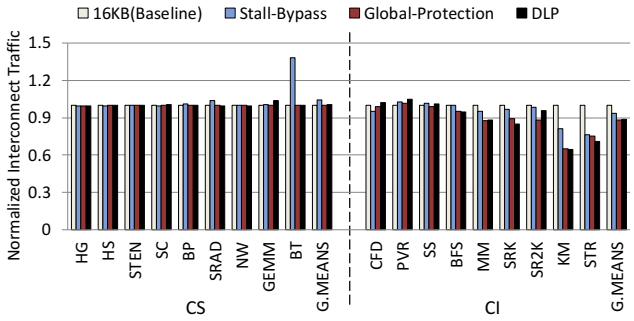


Figure 13: The normalized interconnect traffic with applying the baseline configuration, Stall-Bypass, Global-Protection, and DLP on the 16KB L1D cache.

the effect these three schemes have on L1D cache hits in detail, the normalized numbers of cache hits are plotted in Figure 12b.

6.3.1 CS Applications. Although the numbers of hits for the HG, HS, and STEN applications vary significantly, the L1D cache hits have trivial impact on performance; these applications are dominated by compulsory misses and have a low L1D cache hit rate, as shown in Figure 12b. For the SRAD, NW, and BT applications, Global-Protection and DLP achieve an amount of L1D cache hits similar to the baseline configuration; however, Stall-Bypass fails to retain cache hits for these applications. Considering the relatively high hit rate of SRAD and BT, the loss of cache hits in these two applications can lead to overall performance penalties. Note that the hit rate of Stall-Bypass in SRAD increases in Figure 12a, since the bypassed memory accesses do not count towards the L1D cache rate. In summary, Global-Protection and DLP can preserve more cache hits for applications with many short RDs when compared to Stall-Bypass.

6.3.2 CI Applications. The amount of hits of Global-Protection and DLP exceeds that of Stall-Bypass on most CI applications, with the exceptions of CFD, PVR, and SS. In CFD and PVR, the DLP scheme has less cache hits even compared to the baseline configuration. For other CI applications, Global-Protection and DLP outperform Stall-Bypass in both the number of L1D cache hits and the L1D cache hit rate. Although DLP does not always have more L1D cache hits than Global-Protection and Stall-Bypass, it always has less cache traffic for CI applications; it turns out that the hit rate of DLP is always higher, as shown in Figure 12a. The higher hit rate of DLP can reduce L1D cache stall time, which eventually contributes to the improvement of overall performance.

6.4 Impact on Interconnect Traffic

In order to capture data locality and reduce L1D cache traffic, Stall-Bypass, Global-Protection, and DLP bypass some memory accesses, which will consequentially affect the interconnect traffic from the L1D to the L2 cache. It should be noted that bypassing cache traffic to the interconnect network can be potentially favorable or unfavorable for overall performance. On one hand, bypassing cache misses can reduce the number of eviction writes to the L2 cache. On the other hand, memory accesses that bring in reusable cache lines could be bypassed, which may result in fewer cache hits in the L1D cache and more memory requests in the interconnect network in the future. Figure 13 depicts the normalized interconnect traffic generated by the baseline configuration, Stall-Bypass, Global-Protection, and DLP. For CS applications, both Global-Protection and DLP have a negligible impact on interconnect traffic. However, Stall-Bypass has considerably more traffic for applications like BT, as it over-bypasses memory accesses and loses many reusable cache lines that could have been brought into the L1D cache. For CI applications, Global-Protection and DLP have a greater impact on reducing interconnect traffic than Stall-Bypass and baseline configuration. DLP outperforms other schemes by both bypassing more L1D cache misses (shown in Figure 11a) and capturing more L1D cache hits (shown in Figure 12b). The average decrease in interconnect traffic is 6.2% with Stall-Bypass and 11.5% with DLP. The degree of interconnect traffic reduction is much less than L1D cache traffic because the interconnect network also serves other L1 caches like L1I, L1C, and L1T.

7 RELATED WORK

7.1 Cache Management Policies in CPUs

Our work is inspired by Duong et al. [7], which describes the RDs of a CPU application using statistical techniques. By selecting a threshold slightly higher than most RDs, they can identify a PD that extends the cache line's lifetime to accommodate most RDs in the application. They devise a Protecting Distance-based Policy (PDP) to calculate this global PD. The PDP is optimized for maximizing the cache hit rate based on a RDD collected at runtime. As Section 3.3 mentioned, the PDP is designed for a single-core CPU platform, and cannot accommodate diverse reuse patterns in GPU applications. Many other cache management policies have been proposed to reduce cache thrashing in CPUs [12, 15]. These policies cannot be applied to GPU's L1D caches due to their capacity constraint.

7.2 Alleviating Cache Thrashing in GPUs

Jia et al. [13] characterize performance of applications on GPUs with caches, providing a taxonomy for reasoning about different types of access patterns and locality. They find that the interconnect traffic from the L1D to L2 cache increases due to L1D cache thrashing behavior for certain GPU applications, eventually harming the overall IPC performance. Many works focus on addressing this problem; we classify these works into two major categories.

Since cache thrashing is caused by many memory accesses from parallel GPU threads, reducing the number of parallel GPU threads is an option to alleviate this behavior. Rogers et al. [24] devise a Cache-Conscious Warp Scheduling (CCWS) algorithm. This algorithm also uses the VTA to collect locality loss statuses for building a scoring system to track locality loss for each warp. If certain warps lose many hits because of line replacements, the system will throttle other warps to reduce their locality loss. They show that their mechanism can improve the performance for highly cache-sensitive workloads. Other works like [8, 14, 16] focus on limiting the number of concurrent threads inside GPU cores. The major concern with controlling thread parallelism is that it completely stops parts of threads, potentially under-utilizing the memory bandwidth.

Another important strategy for mitigating cache thrashing behavior is to bypass some memory accesses based on cache information. Li et al. [18] separate tags from data cache. By using a larger tag array and applying the Least Frequent Used replacement policy instead of the LRU replacement policy in the tag array, they can collect the reuse frequency of the data lines; this allows them to bypass memory accesses for lines with low-frequency usage. Since they collect reuse information based on individual memory addresses and GPU applications typically consume a large memory space, it is challenging to predict each line's reuse pattern accurately with a small tag array. To aggregate more information, Tian et al. [26] classify cache access information based on memory instructions. They propose a design with a PC-based table, where each entry of the table uses a saturating counter to record the access history of a memory instruction. If the saturating counter of a specific memory instruction passes a threshold, cache accesses from the this memory instruction will be bypassed. Lee et al. [17] improve the method proposed by [26] through bypassing only some cache accesses from a certain memory instruction instead of all of them.

There are several works putting efforts towards combining both the thread-limiting and the L1D-bypassing techniques. Xie et al. [27] utilize the compiler to determine which load instructions to bypass. Furthermore, they also throttle memory accesses from certain thread blocks based on runtime information. Chen et al. [6] integrate PDP [7] with CCWS [24], which tunes both the PD and the number of warps to keep interconnect traffic in a desired range.

8 CONCLUSION

In this work, we demonstrate that the L1D cache size is not sufficient for many GPU applications, resulting in many L1D cache lines getting replaced before they are re-referenced. By examining the cache access patterns of these applications, we determine that a L1D cache with low associativity cannot retain data locality to satisfy their diverse reuse behaviors; this leads to frequent line replacements with low data re-usage. To improve the efficiency of the L1D cache, we design a DLP scheme to preserve valuable cache lines and increase data locality utilization. DLP predicts protection distances for each memory instruction at runtime, which helps keep a balance between data locality exploitation and over-protection for cache lines with longer reuse distances. Our evaluation shows that DLP can reduce the cache congestion and improve overall IPC performance for Cache Insufficient applications.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF) under award CCF-1149539.

REFERENCES

[1] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. of the 2009 IEEE Int. Symp. on Performance Analysis of Systems and Software*. 163–174.

[2] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*. Springer, 244–263.

[3] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. 2008. Fast support vector machine training and classification on graphics processors. In *Proc. of the 25th Int. Conf. on Machine Learning*. 104–111.

[4] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. 2008. A map reduce framework for programming graphics processors. In *the 3rd Workshop on Software Tools for MultiCore Syst.*

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Sokadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of the 2009 IEEE Int. Symp. on Workload Characterization*. 44–54.

[6] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient gpu computing. In *Proc. of the 47th annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE, 343–355.

[7] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proc. of the 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE, 389–400.

[8] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proc. of the 38th Annual Int. Symp. on Computer Architecture*. 235–246.

[9] Peter N Glaskowsky. 2009. NVIDIA's Fermi: the first complete GPU computing architecture. White Paper. (2009). http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_Nvidia's_Fermi-The_First_Complete_GPU_Architecture.pdf

[10] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proc. of the 2012 Innovative Parallel Computing*. IEEE, 1–10.

[11] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. 2008. Mars: a MapReduce framework on graphics processors. In *Proc. of the 17th Int. Conf. on Parallel Architectures and Compilation Techniques*. ACM, 260–269.

[12] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 60–71.

[13] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proc. of the 26th ACM Int. Conf. on Supercomputing*. ACM, 15–24.

[14] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proc. of the 22nd Int. Conf. on Parallel Architectures and Compilation Techniques*.

[15] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. 2010. Sampling dead block prediction for last-level caches. In *Proc. of the 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE, 175–186.

[16] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proc. of 20th IEEE Int. Symp. on High Performance Computer Architecture*. 260–271.

[17] Shin-Ying Lee and Carole-Jean Wu. 2016. Ctrl-C: Instruction-Aware Control Loop Based Adaptive Cache Bypassing for GPUs. In *Proc. of The 34th Int. Conf. on Computer Design*. IEEE, 133–140.

[18] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proc. of the 29th ACM Int. Conf. on Supercomputing*. ACM, 67–77.

[19] John Nickolls and William J Dally. 2010. The GPU computing era. *IEEE Micro Magazine* 30, 2 (2010), 56–69.

[20] Nvidia. 2011. Tesla M2090 Dual-Slot Computing Processor Module. White Paper. (2011). http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf

[21] Nvidia. 2012. NVIDIA Kepler GK110 Architecture Whitepaper. White Paper. (2012). http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[22] Nvidia. 2017. Developer Zone. (2017). https://developer.nvidia.com

[23] Nvidia. 2017. Nvidia CUDA Samples. (2017). http://docs.nvidia.com/cuda/cuda-samples/index.html

[24] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proc. of the 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE, 72–83.

[25] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[26] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU Cache Bypassing. In *Proc. of the 8th Workshop on General Purpose Processing Using GPUs*. ACM, 25–35.

[27] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Proc. of the 21st Int. Symp. on High Performance Computer Architecture*. IEEE, 76–88.