Compiler-Assisted GPU Thread Throttling for Reduced Cache Contention

Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, Hwansoo Han
Sungkyunkwan University
Suwon, Korea
{hjunkim,sungin.h,hyunsu,euiseong,hhan}@skku.edu

ABSTRACT

Modern GPUs concurrently deploy thousands of threads to maximize thread level parallelism (TLP) for performance. For some applications, however, maximized TLP leads to significant performance degradation, as many concurrent threads compete for the limited amount of the data cache. In this paper, we propose a compiler-assisted thread throttling scheme, which limits the number of active thread groups to reduce cache contention and consequently improve the performance. A few dynamic thread throttling schemes have been proposed to alleviate cache contention by monitoring the cache behavior, but they often fail to provide timely responses to the dynamic changes in the cache behavior, as they adjust the parallelism afterwards in response to the monitored behavior. Our thread throttling scheme relies on compile-time adjustment of active thread groups to fit their memory footprints to the L1D capacity. We evaluated the proposed scheme with GPU programs that suffer from cache contention. Our approach improved the performance of original programs by 42.96% on average, and this is 8.97% performance boost in comparison to the static thread throttling schemes.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → *Compilers.*

KEYWORDS

GPGPU, Static Analysis, Thread Throttling, Cache Contention

ACM Reference Format:
Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, Hwansoo Han. 2019. Compiler-Assisted GPU Thread Throttling for Reduced Cache Contention. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337886

1 INTRODUCTION

Graphics processing units (GPUs) have become the favored accelerators in a variety of domains, ranging from embedded systems to high-performance computing. This popularity comes from the GPU's capability to exploit massive thread level parallelism (TLP) and maximize the performance. On the other hand, the caches in the memory hierarchy are manufactured with the same level of fabrication technology as CPUs. Thus, compared to CPUs, the per-thread capacity of the data cache in GPUs is insufficiently small [6, 10, 12, 13, 18, 28, 30, 31, 37, 38]. The state-of-the-art *Nvidia Volta* GPU supports up to 64 concurrent warps on each streaming multiprocessor (SM), and these warps share a 32KB – 128KB L1 data cache (L1D). When 32 threads in a warp request memory accesses with a high locality, those requests are coalesced into a few memory transactions and fetched into a small number of cache lines. However, the L1D footprints can increase up to 32 cache lines, when 32 memory accesses in a warp cannot be coalesced at all — this is called *memory divergence* [29, 32–34, 36]. If a memory access instruction within a loop incurs memory divergence across many concurrent threads, cache thrashing is inevitable due to the limited L1D capacity. In such cases, even data locality within a thread cannot be exploited at all by the L1D cache. Furthermore, cache thrashing persists throughout the whole execution of the loop.

To address this, thread throttling has been proposed to increase the cache hit rate by limiting a group of threads from sharing the L1D simultaneously. For example, cache-conscious wavefront scheduling (CCWS) [28] limits the number of active warps. Dynamic CTA scheduling (DYNCTA) [13] limits the number of active thread blocks (TB, a group of warps). However, these prior thread throttling schemes commonly require the cache monitoring unit equipped in the GPU hardware to predict cache thrashing and dynamically throttle the TLP at run time. CCWS and DYNCTA also require the inter-phase cost before the adjustment in thread throttling, since they need to detect the changes in the loss of locality. Consequently, the best thread throttling chosen statically for each kernel and each loop outperforms dynamic thread throttling, as the dynamic approach is often too coarse to make fine-granular decisions for applications with fluctuating cache contention [28–30, 37].

In this paper, we present a *compiler-assisted thread throttling* (CATT) that overcomes the drawbacks of prior thread throttling schemes. At compile time, CATT statically analyzes the L1D footprints from the loops showing data reuses across loop iterations. Then, it embeds a thread throttling code which will limit the concurrent execution of active warps or TBs during run time. The throttling code is calibrated to make the cache footprints of a loop fit into the L1D capacity. The cache contention estimation based on static analysis of GPU programs introduces lots of technical challenges. Our static analysis needs to determine the L1D footprint generated by off-chip memory instructions in a loop and analyze the data reuse across iterations by examining the array index. Most of the data used in GPU computing are arrays stored in off-chip

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337886

memory. The expressions used in the array indexes are often complex and involving thread identifiers and loop indexes. Still, they are typically integer linear equations. Thus, the L1D cache footprints can be calculated at compile time. CATT introduces a simple yet effective method to analyze off-chip memory requests in a loop. Taking memory request coalescing and reuse distance across iterations into consideration, CATT can calculate an efficient thread throttling factor for each loop. This effectively limits the number of active thread groups and alleviates cache contention in GPU programs, consequently. By doing so, CATT achieves a precise level of thread throttling at compile time and finds an effective way to the GPU applications with dynamically fluctuating cache contention. In addition, it is a solely software based scheme for GPU compilers and applicable to current GPU systems without hardware modifications.

In summary, we propose a compiler optimization scheme to reduce cache contention in GPU programs. We demonstrate our scheme, CATT, statically analyzes cache contention for well-established GPU benchmarks. We also introduce software-based code transformation for thread throttling, which effectively restricts the number of active thread groups without any hardware modification. With a through experimental evaluation on an Nvidia Volta architecture, we found our scheme successfully improved the performance for GPU applications with dynamically fluctuating cache contention.

The remainder of this paper is organized as follows. Section 2 introduces the background of GPU architecture, and it also describes related work. Section 3 discusses cache contention in GPU computing. Section 4 describes our compiler-assisted thread throttling. Section 5 presents experimental results. Finally, Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 GPU Architecture

GPUs concurrently execute thousands of threads to achieve massive thread level parallelism (TLP), operating in single-instruction-multiple-threads (SIMT) architecture. A warp consists of 32 threads and multiple warps are grouped into a programmer-defined thread block (TB). Every thread has a unique identifier in a GPU program, which enables each thread to process different data. A GPU application has multiple kernels and each kernel is executed with multiple TBs. When a kernel is launched, every TB is distributed among streaming multiprocessors (SMs). Each SM has multiple warp schedulers to issue warps to the CUDA cores. In recent Nvidia GPUs, an SM has 64 CUDA cores, and each runs two warps simultaneously and up to 64 active warps concurrently [24].

On-chip memory in a GPU is composed of register file, multi-level caches (L1 and L2 caches), and shared memory. Register file is partitioned and distributed over active threads. Thus, each thread has its own private set of registers for fast context switches. Each SM has a single on-chip cache memory that is shared by L1 cache and shared memory. In recent Nvidia GPUs, the single on-chip cache memory can be allocated to shared memory (0KB – 96KB) based on programmer configuration, and the remaining are allocated as L1D (32KB – 128KB) at compile time [24]. Shared memory is explicitly managed in source code written by programmers. A large portion of shared memory is often unused, as programmers prefer L1D cache to shared memory for programming simplicity [8, 37]. Every TB running on an SM has private address space in shared memory. Thus, TLP may decrease, when a single TB allocates and uses a large amount of space in shared memory [14, 17]. Furthermore, the register file usage per thread can also reduce the maximum number of concurrent TBs running on an SM.

Table 1: GPU specifications for Nvidia Titan V GPU

GPU	Titan V
Architecture	Volta
SMs	80
Register file / SM	256 KB
L1 cache / SM	32-128 KB
Shared memory / SM	0-96 KB
L2 cache / SM	4608 KB

Table 1 shows the specifications of Nvidia Titan V GPU used in our experiments. However, our scheme is not only limited to the Nvidia GPU architectures but also applicable to other GPU architectures. Titan V employs the Volta architecture, and 5,120 CUDA cores are evenly distributed among 80 SMs in it. Each SM contains 32 load/store units, and 80 SMs can issue up to 2,560 memory requests per cycle [3]. In Titan V, 128KB on-chip memory is shared between L1D cache and shared memory [24]. The size of the L1D and shared memory is configured at compile time. Other GPU architectures, such as Maxwell and Pascal, have a separate address space for the L1D and shared memory. Thus, both on-chip memories have the fixed capacities.

2.2 Cache Contention Reduction

To reduce cache contention, several thread throttling schemes have been proposed [6, 13, 28–30, 37]. CCWS [28] is a dynamic thread throttling scheme which limits the number of active warps at run time based on monitored cache behavior. It reduces the cache contention and improves cache hit rate as a result, but requires hardware modification to monitor the cache and adjust the warp scheduling. Dynamic thread throttling requires warm-up and cool-down time to detect cache thrashing and adjust to the optimal level of thread throttling. To address this challenge, they also proposed static warp limiting (Best-SWL), which finds the best performing case after executing all possible TLPs for an application. Best-SWL often outperforms dynamic thread throttling (CCWS), as optimal TLPs are determined at compile time without monitoring and transition periods [28–30, 37]. Best-SWL has a low implementation overhead but requires programmers effort to find the optimal active warp count by running all possible warp counts. It provides a fixed number of concurrent warps throughout the execution of an application. Thus, it may fail to select the optimal TLPs, when cache contention is dynamically fluctuating during execution [23, 37]. The larger and more diverse the application is, the less likely CCWS and Best-SWL will capture peak performance. On the other hand, CATT statically estimates the degree of cache contention and applies thread throttling to eliminate cache thrashing in GPU programs at compile time. Since CATT can make finer-granular decisions than

Table 2: GPGPU workload description

Abbr.	Application	SMEM (KB)	Input
CS (Cache Sensitive Applications) group			
GSMV [7]	Scalar, vector matrix multiplication	0	20K×20K
SYR2K [7]	Symmetric rank-2k operations	0	2K×2K
ATAX [7]	Matrix transpose and vector mul.	0	40K×40K
BICG [7]	BiCGStab linear solver	0	40K×40K
MVT [7]	Matrix vector product and transpose	0	40K×40K
CORR [7]	Correlation computation	0	2K×2K
BFS [4]	Breadth-First search	0	graph128k.txt
CFD [4]	CFD solver	0	missile.domm.0.2.M
KM [4]	Kmeans	0	819200.txt
PF [4]	Particle filter	4.00	128×128×10
CI (Cache Insensitive Applications) group			
GRAM [7]	Gram-Schmidt process	0	2K×2K
SYRK [7]	Symmetric rank-k operations	0	1K×1K
BT [4]	B+ tree	0	mil.txt, command.txt
HP [4]	Hotspot3d	0	512×8
LVMD [4]	LavaMD	7.03	boxes1d 10
2MM [7]	2 matrix multiply	0	1K×1K
GEMM [7]	Matrix multiply	0	0.5K×0.5K
3MM [7]	3 matrix multiply	0	0.5K×0.5K
BP [4]	Back propagation	1.06	64K
HM [4]	Huffman	6.13	test1024
LUD [4]	LU decomposition	6.00	256
HW [4]	Heart wall	11.59	test.avi
MC [4]	Myocyte	0	100
NW [4]	Needleman-Wunsch	4.25	2K
PFR [4]	Pathfinder	2.00	100K
SC [4]	Streamcluster	0	64K

Best-SWL, our approach is effectively applicable to programs with various phases of cache contention.

Divergence-aware warp scheduling (DAWS) [29] improved over CCWS by adjusting thread throttling proactively. It samples a few warps to detect memory divergent accesses and predicts the required cache footprint for currently running warps. If a new warp incurs cache contention, DAWS stops the new warp to run for thread throttling. It works well on applications with irregular access patterns, better than Best-SWL. Since Best-SWL provides a fixed level of thread throttling throughout the whole execution, it may be sub-optimal in some portion of kernels. On the other hand, DAWS adapts schedule not to incur cache contention for currently running warps with a good intra-thread/warp locality. Our approach statically attempts to find a fine granular thread throttling for individual loops, which can result in similar performance to DAWS on regular access patterns. However, DAWS performs better on irregular access patterns than ours. Still, the benefit of our CATT scheme is that our compiler-based optimization is readily applicable to current GPU architectures without hardware changes.

DYNCTA [13] is another dynamic thread throttling for GPUs, which modifies hardware modules to monitor the idle cycles of the computational unit as well as memory units at run time. Since DYNCTA limits the number of concurrent threads at TB granularity, the performance improvement is relatively insignificant compared to CCWS. CIAO [37] also requires several hardware modifications

```
1  #define NX 40960
2  // L1 cache size: 32KB, shared memory size: 96KB
3  // atax_kernel1<<<80*4, 256>>>(A, B, tmp)
4  __global__ void atax_kernel1(float *A, float *B, float *tmp) {
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6      if (i < NX) {
7          for(int j=0; j < NX; j++) {
8              tmp[i] += A[i * NX + j] * B[j];
9          }
10     }
11 }
```

Figure 1: GPU kernel example for cache contention

to reduce cache contention. To eliminate cache thrashing, they first redirected warps causing severe interference at the L1D cache to unused shared memory. Then, they throttle such warps when they still cause interference. CIAO requires many changes in the memory system, as it needs to create a direct path to shared memory from the L2 cache. CIAO also needs a new data structure to use the space of shared memory as an extended area of the L1D cache. Previous dynamic thread throttling methods (CCWS, DAWS, and DYNCTA) were implemented and evaluated in the GPU simulator. However, our compiler-based thread throttling, CATT, is a pure software-based method that can be evaluated on real GPU devices.

Yanhao et al. [6] first introduced software thread throttling. They target irregular applications operating on sparse data where one thread repeatedly visits neighboring nodes. They divide the entire workloads into multiple partitions so that the working set of each partition fits into the L1D while keeping the data communications among different partitions minimum. Then, they process data-balanced partitions independently with the application-level scheduling policies. Our work primarily focuses on cache locality in any type of GPU applications whereas their approach focuses on irregular sparse matrix applications. Our approach applies a few typical code transformations to the original code by the compiler, which is more general for many regular applications.

Several schemes were also proposed to solve the cache contention by selectively bypass the cache [3, 11, 16, 20, 21, 23, 35, 36]. Several methods have been proposed to select instructions and memory requests for bypassing caches on cache efficiency and access locality. In GPU computing, bypassing the L1D cache for threads or warps shows significant effectiveness in some cases because of its massive scale of TLP. However, the cache bypassing cannot prevent loss of locality for threads or instructions with cache locality that bypass the L1D cache [5].

2.3 Memory Access Analysis

Several works analyzed and optimized memory access patterns in GPU programs [1, 2, 9, 15, 19, 22]. Through the analysis, these works have successfully identified inefficient performance factors in GPU's memory system. Kim et al. [15] proposed a method to estimate the memory performance of GPU programs. It enables programmers to optimize memory accesses by effectively using both shared and global memory. Li et al. [22] also introduced a scheme to revise the placement of data between different types of on-chip memories. Alur et al. [1, 2] proposed a scheme that identifies

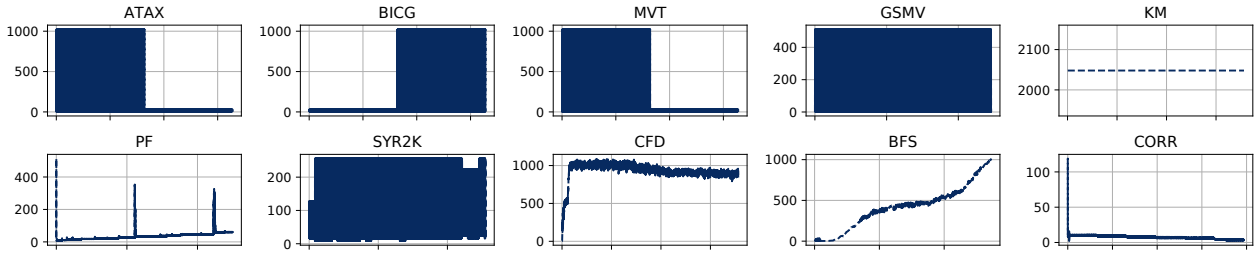

Figure 2: Number of off-chip memory requests (after coalescing) over time in the CS applications (X-axis: off-chip memory instruction sequence, Y-axis: memory requests per instruction)

uncoalesced memory access bugs in GPU programs. They also presented another method for deciding block-size independence in GPU code. They investigated expressions in memory accesses, particularly array indexes, to detect uncoalesced memory accesses. Our static analysis is similar in investigating array indexes, but our analysis determines the degrees of cache contention and decide thread throttling factors.

3 CACHE CONTENTION IN GPU COMPUTING

As stated earlier, the cache memory of a GPU is scarce resource because the size of per-thread cache is much smaller than that of a CPU. When a large number of threads accesses the off-chip memory, *inter-thread locality* becomes as much important as *intra-thread locality*. Since a memory instruction in SIMT model may generate memory requests as many as the number of actively running threads, locality among those requests should be good enough to coalesce many requests into a small number of memory addresses and then cache lines. One memory instruction with a poor inter-thread locality may easily pollute a large portion of the data cache. In such cases, intra-thread locality in the kernel code can not be exploited at all because the cache lines allocated for the other threads will be evicted by the thread before they are reused. This is cache contention among threads, which greatly degrades the performance of GPU applications. In order to find out benchmark applications with potential cache contention, we ran the applications on the two different L1D cache configurations — 64KB and up to 128KB and categorized them into two groups — cache-sensitive (CS) and cache-insensitive (CI) based on their observed behavior [16]. Where application required shared memory, the maximum cache size was set to 96KB. Based on the experimental results, applications are cache-sensitive when their L1D hit rate increases over 10% in a larger cache than 64KB. Otherwise, they are cache-insensitive and tend not to experience cache contention. Through the analysis, we observed that the CI applications have no cache reuse or resolved cache contention with a large cache.

Table 2 lists benchmark applications from Rodinia [4] and Polybench/GPU [7] in two groups. In a statement outside the loop body, the working set itself tends not to be large enough to reach a certain degree of cache contention. We consider only the loop body for optimization and exclude the applications that do not contain a loop body (*i.e.*, 2dconv, 3dconv, fdtd-2d, gaussian, nn, and srad2). The table also shows the size of shared memory (*SMEM*) and input parameters used by benchmark applications.

3.1 Cache Contention Example

Figure 1 is an example code for cache contention. Three off-chip memory accesses — $tmp[i]$, $A[i * NX + j]$, and $B[j]$ are executed repeatedly in a loop. The variable i is a linearized thread identifier assigned outside the loop. Naturally, this value remains as a constant within a thread context, and it varies across different threads. Since the coefficient of linearized thread id (i) in the index expression of $tmp[i]$ is 1, the address distance among threads, which is called *inter-thread distance*, is 1. This means these memory accesses have inter-thread locality. Meanwhile, the index expression in $tmp[i]$ is constant in a thread, which results in intra-thread locality with the same address across loop iterations. This address distance within a thread is called *intra-thread distance* and it is 0 for this case. As for $B[j]$, the index expression is constant throughout the whole threads, which leads to inter-thread locality with inter-thread distance of 0. Within a thread context, the coefficient of iterator variable (j) is 1, which again leads to intra-thread locality with intra-thread distance of 1 across loop iterations.

While the two array accesses show both locality, the index expression in $A[i * NX + j]$ does not show any locality. It has the coefficient NX for the linearized thread id (i), which means each thread accesses memory address apart from each other by NX inter-thread distance. As NX is a large constant defined as 40960, two memory addresses for two adjacent threads are 40960 * 4 byte apart. Since this distance is far bigger than the size of cache line, these accesses have no inter-thread locality. Within a thread, the linearized thread id (i) and its coefficient (NX) are constant. Thus, the coefficient of iterator variable determines intra-thread locality. As for $A[i * NX + j]$, the coefficient of iterator variable j is 1 and this results in intra-thread locality with intra-thread distance of 1 across loop iterations. Since memory accesses from $tmp[i]$ and $B[j]$ have both inter-/intra-thread locality, they are well coalesced and fetched into a small number of cache lines. However, memory accesses from $A[i * NX + j]$ show poor inter-thread locality, which will issue too many uncoalesced off-chip requests and pollute the L1D cache as a result. This pollution is not only for array A but also intervene accesses for array tmp and array B. Thus, all threads contend each other for the cache capacity without exploiting potential locality.

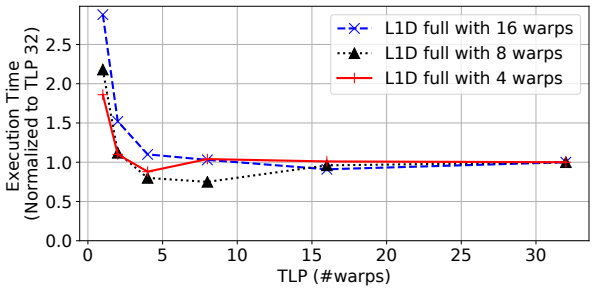

Figure 3: Performance impact on TLP and cache footprints

3.2 Dynamic Changes in Cache Contention

Figure 2 plots the number of off-chip memory requests over time for the benchmark applications in the CS group. The access requests produced by off-chip memory instructions are counted after memory coalescing. Thus, the higher number implies the more divergent in memory accesses and the lower number implies memory requests are well coalesced. The more the divergence of memory accesses are, the more likely the cache contention occur. For example, ATAX exhibits two contrasting execution phases. The first phase is highly memory-divergent while the second phase is well memory-coalesced. Cache contention occurs in the first phase where thread throttling is required to reduce cache contention. Meanwhile, the second phase does not experience cache contention and the high level of thread-level parallelism helps improve the performance during the second phase. Similarly, BICG, MVT, PF, CFD, BFS, and CORR show multiple distinct execution phases. As the cache contention does not persist throughout the whole execution of the benchmark applications, thread throttling should be applied only to a certain period of time within a program, even within a kernel. In this context, our CATT scheme can selectively apply thread throttling to individual loops in a kernel. With precise analysis on cache footprints, the compile-time transformed code by CATT timely limits the number of active thread groups at run time.

3.3 Trade-offs between TLP and Cache Footprints

Figure 3 plots the performance impact on the varying levels of TLPs. Each line represents a microbenchmark that fills the L1D capacity with a certain number of warps. For instance, *L1D-full-with-8-warps* represents a microbenchmark where the footprints generated by eight concurrent warps fill the L1D capacity. Thus, if we increase the TLP to 16 or 32, the memory footprints exceed the L1D capacity, which incurs cache contention among threads according to our observation. As shown in Figure 3, execution times at 16 and 32 TLPs are higher than that of 8 TLPs, mainly due to cache contention. Meanwhile, if we decrease the TLP to 1, 2, or 4, the memory footprints are still within the L1D capacity, but we underutilize the GPU's thread parallelism. As a result, the execution times at 1, 2, and 4 TLPs are higher than that of 8 TLP, due to low utilization of GPU. Thus, the proper level of thread parallelism should be the highest possible TLP, but not exceed the L1D capacity with its memory footprint. Similar interpretations hold for other microbenchmarks — *L1D-full-with-4-warps* and *L1D-full-with-16-warps*. With this experiment, We find that footprints in the L1D must fit into the L1D capacity to avoid cache contention. To enhance the performance of GPU applications, we devise a scheme to throttle certain number of thread groups and make the memory footprint fit into the L1D capacity.

4 COMPILER-ASSISTED THREAD THROTTLING

Our thread throttling scheme, CATT utilizes two techniques for code transformations for thread throttling. The degree of thread throttling is determined based on the compile-time estimation of cache contention. First, we need to determine the largest possible cache size that maximizes the TLP. Since the size of shared memory limits the degree of TLP, the size configuration of L1D cache and shared memory should be considered. This can be easily achievable because the number of concurrent TBs on an SM is known at compile time. Then, we estimate cache contention for each loop. We track expressions, particularly in the array index, as coefficients of thread index (*tid*) and iterator variables are basic information for the estimation of L1D footprints. Further, our static analysis computes thread throttling factors that effectively reduce the L1D footprints to fit into the L1D capacity. Lastly, we throttle the concurrent threads (warps or TBs) by transforming the source code. We have implemented the static analyzer and the source-to-source compiler using Antlr's C parser [27].

The static analysis based on array indexes is suitable to GPU programs because typical GPU kernels have regular memory access patterns [26, 37].

4.1 Configuring L1D and Shared Memory Sizes

We now configure the capacity of the L1D and shared memory. Programmers can configure the size of the L1D and shared memory at compile time. As the L1D capacity decreases, the degree of cache contention in the CS applications rises. Considering the usage of shared memory, we maximize the L1D capacity. The number of concurrent TBs on an SM decreases, as the usage of register file or shared memory in a TB increases. For instance, each SM has a limited register file and shared memory capacity and TBs deployed on an SM share these memory spaces. The excessive use of the register file or shared memory by a TB can lead to TLP reduction.

Equation 1 and Equation 2 are formulas for calculating the maximum number of concurrent TBs on an SM [25]. Each considers the two limiting factors: the use of shared memory and the use of register file. Equation 1 divides the shared memory capacity of an SM by the size of shared memory used in a single TB.

$$\#TB_{shm} = SIZE_{shm\_SM} / USE_{shm\_TB} \quad (1)$$

Equation 1 presents the number of concurrent TBs on an SM restricted by the fact that threads are sharing the limited size of shared memory. Equation 2 divides the register file capacity of an SM by the size of the register file used by a single TB.

$$\#TB_{reg} = SIZE_{reg\_SM} / USE_{reg\_TB} \quad (2)$$

Equation 2 is the number of concurrent TBs on an SM restricted by the fact that threads are sharing the limited size of register file. The use of register file and shared memory is known at compile time

with *Nvcc* compiler's flag *-v*. The amount of shared memory usage is also explicitly declared in the source code.

In the recent GPU architecture, each SM is capable of deploying up to 64 warps simultaneously ($\#TB_{HW}$). The number of TBs that each SM can issue concurrently is the minimum value of $\#TB_{shm}$, $\#TB_{reg}$, and $\#TB_{HW}$. Equation 3 considers the use of register file and shared memory as well as hardware limitations to calculate the number of concurrent TBs on an SM.

$$\#TB_{SM} = Min(\#TB_{shm}, \#TB_{reg}, \#TB_{HW}) \tag{3}$$

Equation 4 computes the minimum capacity of shared memory required for the concurrent TBs on an SM.

$$USE_{shm\_SM} = \lceil USE_{shm\_TB} * \#TB_{SM} \rceil \tag{4}$$

This equation computes the shared memory usage of all concurrent TBs by multiplying the number of concurrent TBs on an SM ($\#TB_{sm}$) by the amount of shared memory used by a single TB. The Nvidia Volta GPU can configure the size of shared memory to be 0, 8, 16, 32, 64, or 96 KB per SM [24]. We choose shared memory capacity for the smallest configurable option that is greater than or equal to USE_{shm_SM} so as to maximize the TLP under given conditions.

4.2 Determining Thread Throttling Factors

The thread throttling factor is a value that restricts the degree of thread groups concurrently sharing the L1D. Our goal is to find a proper throttling factor that makes the L1D footprints fit into the L1D capacity. We first determine the cache locality for all off-chip memory accesses executed in a loop. It is essential to track cache locality because cache thrashing occurs for memory access that has lost cache locality. We then measure the cache contention (L1D footprints) for loops where cache locality presents. Lastly, we compute the thread throttling factor that reduces cache contention by adjusting the number of concurrent thread groups on an SM. In order to statically analyze the memory access patterns in kernel code, we investigate the index of linearized arrays on linearized thread grid, which generally takes the form shown in Equation 5.

$$C_{tid} * tid + C_i * i \tag{5}$$

C_{tid} and C_i are coefficients of thread ID (tid) and linearized iterator variable (i), respectively. Those coefficients represent the reuse distance between threads and the one between consecutive iterations, respectively. Equation 6 represents how to determine if cache locality exists in a loop.

$$C_i \leq SIZE_{cache\_line} : \textit{cache locality exists, if true} \tag{6}$$

Cache locality exploited in a loop means that the fetched cache line is re-accessed in the next iteration. We determine the existence of cache locality within the loop by investigating the coefficient of the iterator variable i (C_i). When the intra-thread distance is smaller than the cache line size, the fetched cache line is reused in the following iteration. If the intra-thread distance is larger than the cache line size, the thread will request a new cache line and does not re-access the fetched cache line.

Next, we estimate the degree of cache contention for all loops. The GPU memory unit coalesces memory requests generated by a single warp into cache lines and fetches them from off-chip memory to the L1D. The coefficient of thread ID (C_{tid}) used in the array index indicates the inter-thread distance, which means the number of cache lines requested by a single warp. The maximum number of cache lines that can be requested by a single warp cannot exceed the warp size ($SIZE_{warp}$) because each thread can request a single address from a single instruction, and each warp consists of 32 threads. Equation 8 represents how to calculate the number of memory requests made by all off-chip memory accesses in a loop.

$$REQ_{warp} = \begin{cases} 1, C_{tid} = 0 \\ Min(C_{tid}, SIZE_{warp}), C_{tid} \neq 0 \end{cases} \tag{7}$$

$$SIZE_{req} = \sum_{mem\_insts} REQ_{warp} * (\#Warps_{TB} * \#TB_{SM}) \tag{8}$$

They multiply the number of concurrent warps on an SM ($\#Warps_{TB} * \#TB_{SM}$) by the inter-thread distance (C_{tid}). For instance, if the inter-thread distance is zero, threads are accessing the same address (4 bytes), and the memory requests are coalesced into a single cache line (128 bytes). If the inter-thread distance is eight (32 bytes), every four threads are requesting a single cache line, and memory requests made by a warp are coalesced into eight cache lines.

For irregular memory access patterns (*i.e.*, indirect-memory access), the value of C_{tid} is not constant at compile time. For instance, in BFS [4], each thread traverses from one node in a graph to a neighboring node. The address of the neighboring node is irregular, and the inter-thread distance is constantly changed through iterations. For applications with irregular memory access patterns, it is challenging to measure the degree of the cache contention at compile time. Based on the result of the measurement, the thread throttling method resolves cache contention by limiting certain active thread groups. However, incorrect measurement of cache contention for irregular patterns can unnecessarily reduce TLP. Thus, we conservatively set the value of C_{tid} to one for irregular memory accesses. Our static analysis may not determine the exact degree of cache contention, but it can prevent performance degradation due to excessive thread throttling.

If the $SIZE_{req}$ value is larger than the L1D capacity, we assume there is a probability of cache thrashing. Thread throttling factor is value for limiting the concurrent group of warps or TBs on an SM. We find the thread throttling factor (N and M) in Equation 9, which successfully reduces the L1D footprint to fit into the L1D capacity.

$$SIZE'_{req} = \sum_{mem\_insts} REQ_{warp} * \frac{\#Warps_{TB}}{N} * (\#TB - M) \tag{9}$$

We reduce the number of concurrent warps on an SM to $1/N$ until the $SIZE'_{req}$ is smaller than the L1D capacity. N is a multiple of 2 to partition $\#Warps_{TB}$ evenly. N cannot be larger than $\#Warps_{TB}$. If the $SIZE'_{req}$ ($N = \#Warps_{TB}$) is still larger than the L1D capacity, we decrease $\#TB_{SM}$ by M. M cannot be larger than $\#TB_{SM}$. Thus, the maximum possible thread throttling factor is when N is $\#Warps_{TB}$ and M is $\#TB_{SM}$. If M and N are set to the maximum thread throttling factor, and the $SIZE'_{req}$ is still larger than the L1D capacity, the thread throttling cannot resolve the cache contention. That is, TLP of CORR application is reduced to the minimum, cache contention still exists. Thus, optimization for applications with such characteristic is not taken into account.

Static analysis well estimates cache contention for single-dimensional TBs, but in practice it sometimes fails for multidimensional TBs. It is challenging to track the array indexes for thread ID of multidimensional TB through static analysis. In particular, unaligned memory addresses of multidimensional thread IDs lead to make inaccurate decisions, and it is also tricky to analyze cache contention

```
1  #define NX 40960
2  #define WS 32 // warp size
3  // L1 cache size : 32 KB, shared memory size: 96 KB
4  // atax_kernel1<<<80*4, 256>>>(A, B, tmp)
5  __global__ void atax_kernel1( float *A, float *B, float *tmp) {
6      int i = blockIdx.x * blockDim.x + threadIdx.x;
7      if (i < NX) {
8          if (threadIdx.x/WS >= 0 && threadIdx.x/WS < 4) {
9              for(int j=0; j < NX; j++) {
10                 tmp[i] += A[i * NX + j] * B[j];
11             }
12         } __syncthreads();
13         if (threadIdx.x/WS >= 4 && threadIdx.x/WS < 8) {
14             for(int j=0; j < NX; j++) {
15                 tmp[i] += A[i * NX + j] * B[j];
16             }
17         } __syncthreads();
18     }
19 }
```

Figure 4: GPU kernel example of warp throttling

if a TB is not a multiple of warp size. In fact, this is unusual in GPU programs [25]. We examine every address accessed by each thread in a warp to make the correct estimation for multidimensional TBs (*i.e.*, SYR2K).

4.3 Transforming Code for Thread Throttling

In this section, we reduce cache contention by modifying the source code that causes cache thrashing. Our software thread throttling consists of two approaches: warp-level throttling for limiting the number of active warps in a TB, and TB-level throttling for that of active TBs running simultaneously on an SM.

Figure 4 is an example source code of the warp-level throttling. We split the loop in Figure 1 into the thread throttling factor N. Each warp ID in a TB can be calculated by dividing their thread ID by warp size ($tid/SIZE_{warp}$). Since the warp-level throttling manages thread groups in warp/TB-granularity, there is no control divergence overhead. In the sample kernel, the first warp group (warp ID: 0 – 3) is executed through the conditional statements, and then the remaining warp group (warp ID: 4 – 7) is executed in order. To ensure the order of execution between the two warp groups, *__syncthreads*() works as a barrier so that all warp groups in the same TB will not continue the next instruction until they all reach the line 16. By limiting the execution of a half of the warp running simultaneously on an SM, we could reduce cache contention in the L1D.

Figure 5 is an example source code for the TB-level throttling. We can adjust the number of concurrent TBs on an SM to M TBs by arbitrarily allocating address space in shared memory. Each SM has 96KB of shared memory, and four TBs in an SM share shared memory. To limit concurrent TBs on an SM, we allocate 48KB of shared memory per TB to increase its usage. We add a simple write command to shared memory so that the compiler does not remove the shared memory allocation instruction. The TB-level throttling allows the warps of the first two TBs to be deployed, and the warps of remaining two TBs wait until the kernel execution of the first two TBs is completed. By limiting two concurrent TBs on an SM, we could mitigate cache contention in the L1D.

```
1  #define NX 40960
2  // L1 cache size : 32 KB, shared memory size: 96 KB
3  // atax_kernel1<<<80*4, 256>>>(A, B, tmp)
4  __global__ void atax_kernel1( float *A, float *B, float *tmp) {
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6      __shared__ float dummy_shared[12288]; // 48 KB
7      dummy_shared[threadIdx.x]=0;
8      if (i < NX) {
9          for(int j=0; j < NX; j++) {
10             tmp[i] += A[i * NX + j] * B[j];
11         }
12     }
13 }
```

Figure 5: GPU kernel example of TB throttling

Table 3: TLP per SM for various methods

App.	Kernel	Loops	($\#warps_{TB}$, $\#TBs$)				
			Baseline	32KB L1D		Max. L1D	
				BFTT	CATT	BFTT	CATT
ATAX	#1	1	(8,4)	(4,2)	(1,4)	(4,4)	(4,4)
	#2	2	(8,4)		(8,4)		(8,4)
BICG	#1	1	(8,4)	(4,3)	(8,4)	(4,4)	(8,4)
	#2	2	(8,4)		(1,4)		(4,4)
MVT	#1	1	(8,4)	(4,2)	(1,4)	(4,4)	(4,4)
	#2	2	(8,4)		(8,4)		(8,4)
GSMV	#1	1	(8,2)	(1,2)	(1,2)	(4,2)	(4,2)
SYR2K	#1	1	(8,8)	(4,2)	(4,3)	(4,8)	(4,8)
KM	#1	1	(8,8)	(4,2)	(1,8)	(2,8)	(2,8)
	#2	2	(8,8)		(1,8)		(2,8)
PF	#1	1	(16,3)	(16,3)	(2,3)	(16,4)	(4,3)
		2	(16,3)		(2,3)		(4,3)
		3	(16,3)		(16,3)		(16,3)
	#2	4	(16,4)		(16,4)		(16,4)
	#3	5	(16,4)		(16,4)		(16,4)
	#4	6	(16,4)		(16,4)		(16,4)
BFS	#1	1	(16,4)	(16,4)	(16,4)	(16,4)	(16,4)
	#2	-	(16,4)		(16,4)		(16,4)
CFD	#1	1	(6,10)	(6,10)	(6,10)	(6,10)	(6,10)
	#2	-	(6,10)		(6,10)		(6,10)
	#3	2	(6,10)		(6,10)		(6,10)
	#4	-	(6,10)		(6,10)		(6,10)
CORR	#1	-	(8,1)	(8,1)	(8,1)	(8,1)	(8,1)
	#2	-	(8,1)		(8,1)		(8,1)
	#3	-	(8,1)		(8,1)		(8,1)
	#4	1	(8,1)		(8,1)		(8,1)

There are several constraints on TB-level throttling. First, if there is no free space available in shared memory, the size ratio of shared memory and the L1D must be reconfigured. Next, the TB-level throttling reduces the TLP throughout kernel execution and can adversely affect the execution phase without cache thrashing. Thus, the CATT takes into account the warp-level throttling first, and the

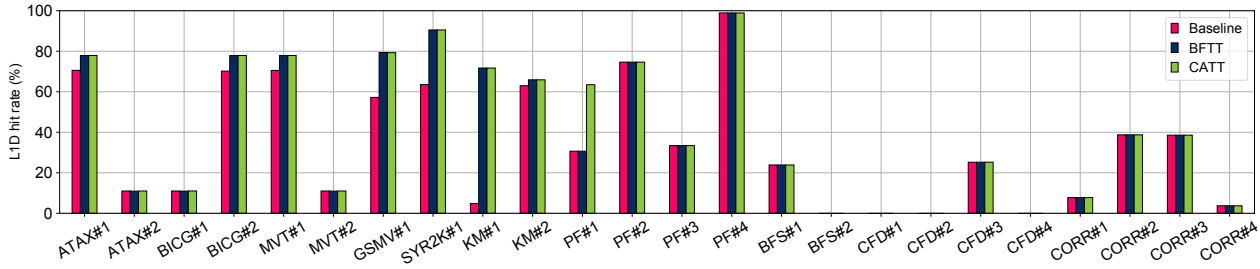

Figure 6: L1D hit rates on maximum L1D cache

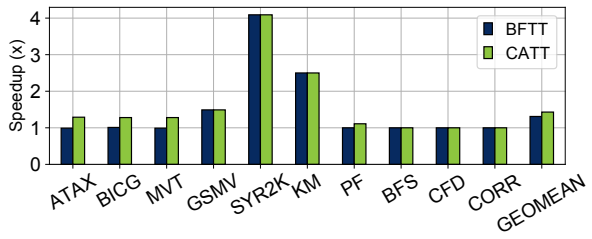

Figure 7: Performance of CS group on maximum L1D cache

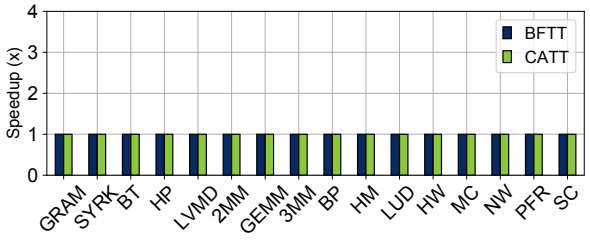

Figure 8: Performance of CI group on maximum L1D cache

TB-level throttling is considered if the $SIZE'_{req}$ is still larger than $SIZE_{L1D}$.

For applications whose kernel function parameters (*i.e.*, grid size, thread block size, shared memory size) are unknown at compile time, the modified kernel function is duplicated with different thread throttling factors. The kernel function is then selectively invoked according to the dynamically determined values. However, we did not observe any such behavior in our experiments.

5 PERFORMANCE EVALUATION

Our experimental evaluation was performed on Nvidia Titan V GPU (Volta architecture) with CUDA toolkit version 10.1 and Nvidia 418.56 GPU driver. Nvidia's *Nvprof* was used for performance measurement. We also used a compile option (*-dlcm=ca*) to support caching global memory in the L1D. We evaluated a large collection of benchmarks from Polybench [7] and Rodinia [4]. We compared our methods with best-fixed thread throttling (BFTT). BFTT attempts to find the best performing case of all possible combinations of concurrent warp counts per TB and TB counts per SM. To throttles threads, BFTT uses warp-level throttling and TB-level throttling methods. In Table 3, the selected TLP ($\#warps_{TB}$, $\#TBs$) via static analysis of CATT is listed along with that of BFTT.

5.1 Performance Analysis

Figure 6 shows the L1D hit rate of CATT and BFTT. Each bar represents a kernel in each application, which is identified by concatenating the application name with kernel numbering (*i.e*, ATAX#1, ATAX#2). Figure 7 shows the execution time of CATT and BFTT, normalized to baseline. For kernels with high cache contention, CATT enhances the cache hit rate. Further, since CATT can have different strategies for each loop, CATT outperforms BFTT for applications with a varying level of cache contention. For example, ATAX, BICG, and MVT applications commonly have multiple kernels, and each kernel has a different level of cache contention. The loop in ATAX#1 kernel originally has a large L1D footprints, thus with high cache contention. CATT and BFTT effectively throttled threads to (4, 4) and reduced the cache contention. However, for the loop in ATAX#2 kernel, cache contention was low. CATT avoids performance degradation by applying TLP of (8, 4). Meanwhile, BFTT degrades performance by applying the same degree (4, 4) of thread throttling as ATAX#1. BFTT throttles threads but does not increase L1D hit rate because ATAX#2 has no cache contention. This is a typical case where CATT performs better than BFTT by setting proper degrees of thread throttling for individual loops. PF has four kernels, and each one has a single loop except PF#1, which has three loops. CATT and BFTT avoid performance degradation by applying the same degree of TLP on loops with low cache contention (PF#2, PF#3 and PF#4). However, we have a different case in the first and the second loop of PF#1, which originally has high cache contention. Reducing the number of concurrent threads will help the performance by eliminating cache contention. CATT efficiently reduces the cache contention by selecting the optimal number (4, 3) of TLP, while BFTT applies the same degree of TLP, which is proper for other loops but too high for the first and the second of PF#1. This makes difference in overall performance of PF, which is shown in Figure 7. Since GSMV, SYR2K, and KM applications have a uniform level of cache contention over execution, CATT and BFTT commonly improved performance to the same degree.

For irregular access patterns, such as BFS and CFD, CATT conservatively estimates cache contention. CATT preserves the original

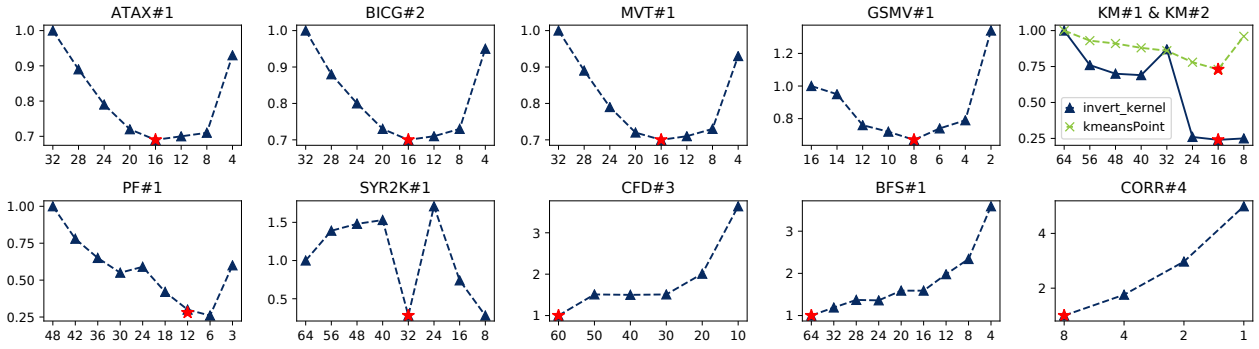

Figure 9: Normalized execution times with various throttling factors of CS group (throttling factors selected by CATT are marked with red stars, X-axis: thread throttling factors ranging from the maximum to the minimum TLP, Y-axis: normalized execution times to the baseline)

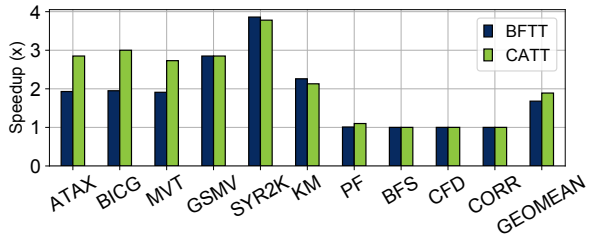

Figure 10: Performance of CS group on 32KB L1D cache

level of TLP not to degrade the performance. CORR has a very high cache contention, but the L1D footprint cannot be reduced to fit the L1D capacity even with the minimum degree of TLP. In such case, kernels and loops need to be split into smaller pieces, which requires algorithm changes in original code. CATT passes such cases without optimization, as cache contention is still high on any degree of TLP. Overall, CATT improved the performance of the baseline by 42.96% on average (geomean), while BFTT by 31.19% for benchmark applications in cache-sensitive group. CATT outperforms BFTT, as it makes a fine-granular decision for each loop in applications. This is typically beneficial, when applications have phase changes in cache contention.

5.1.1 CATT for CI Applications. Figure 8 shows the execution time of CATT and BFTT for benchmark applications in cache-insensitive group. Since CI applications are not sensitive to the L1D capacity, throttling the threads of CI applications not only reduces TLP but also degrades performance. We demonstrated that our method does not mislead cache contention for CI applications. Through the static analysis, CATT successfully estimated the footprint in L1D and selected the correct degree of TLP, which is the same TLP as the baseline. Likewise, BFTT also experimentally selected the same degree of TLP that yields the best performing cases.

5.1.2 Sensitivity to Thread Throttling Factors. Figure 9 plots the normalized execution times on various thread throttling factors from the maximum to the minimum number of concurrent warps per SM. The red star marks are the degrees of thread throttling selected by our scheme, CATT. This evaluates the accuracy of our static analysis in CATT. CATT selects the optimal degrees of thread throttling for applications with regular patterns. However, the optimal decisions for irregular patterns are challenging to make, since their cache contention behaviors are unknown at compile time. The performance of such kernels — PF#1, BFS#1, and CFD#3 are not the best with CATT. For PF#1, the best performance is achieved, when selecting a slightly larger thread throttling factor than CATT. For BFS#1 and CFD#3, the best performance is achieved with CATT. However, in the case of BFS#1 and CFD#3, the degree of cache contention is constantly changing throughout iterations, which requires different throttling strategies for each iteration. Since our thread throttling can make one fixed decision for the entire loop body, this method is not always the optimal solution for this irregular pattern.

5.1.3 Sensitivity to L1D Capacity. Cache contention increases as a large number of concurrent threads compete for small L1D. Thus, L1D capacity reduction often results in significant cache contention for cache sensitive applications. To evaluate the effect of thread throttling on a small L1D, we configured the L1D to 32KB. Since the L1D capacity in previous generations of GPU ranges from 16KB to 48KB, this experiment helps understand the benefits of our thread throttling scheme on older architectures.

The speedup of CATT and BFTT on 32KB L1D are shown in Figure 10, normalized to the baseline. CATT and BFTT improved the performance of the baseline by 89.23% and 68.17% on average (geomean), respectively. The improvements in 32KB L1D are much more significant than L1D with maximum capacity. Since our thread throttling scheme, CATT, is more effective on GPUs with small L1D capacity, it can be effectively applied to GPUs in previous generations or ones in mobile systems.

5.1.4 Analysis Overhead. Static analysis is a considerably fast method that completes the analysis within seconds for most benchmark applications. Our static analysis has an algorithm that is linear to the length of the source code, and the analysis for most applications is completed within 1–2 seconds.

6 CONCLUSION

This paper proposed CATT, a compiler-assisted thread throttling scheme. In the proposed scheme, in order to address thread throttling for dynamically changing cache contention, a fine granular thread throttling is applied to individual loops in GPU kernels. CATT performs cache contention analysis and code transformations for thread throttling at compile time. CATT calculates the cache contention of the L1D based on array index expressions and insert thread throttling code to restrict the thread groups from sharing of the L1D simultaneously. Our software thread throttling requires no hardware modification and is easily applicable to the current GPU system. For 10 GPU applications suffering from cache thrashing, CATT achieved the performance improvement in the Titan V GPU by 42.96% on average, which is 8.97% boost over static thread throttling schemes.

ACKNOWLEDGMENTS

This research is supported in part by the National Research Foundation in Korea under PF Class Heterogeneous High Performance Computer Development NRF-2016M3C4A7952587.

REFERENCES

[1] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2017. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *Computer Aided Verification (CAV)*.
[2] Rajeev Alur, Joseph Devietti, and Nimit Singhania. 2018. Block-Size Independence for GPU Programs. In *Static Analysis (SAS)*.
[3] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. 2015. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *International Conference on Parallel Architecture and Compilation (PACT)*.
[4] 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)*.
[5] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[6] Yanhao Chen, Ari B. Hayes, Chi Zhang, Timothy Salmon, and Eddy Z. Zhang. 2018. Locality-Aware Software Throttling for Sparse Matrix Operation on GPUs. In *2018 USENIX Annual Technical Conference (ATC)*.
[7] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-Tuning a High-Level Language Targeted to GPU Codes. In *Innovative Parallel Computing (InPar)*.
[8] Ari B. Hayes and Eddy Z. Zhang. 2014. Unified On-chip Memory Allocation for SIMT Architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*.
[9] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. 2011. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 105–118.
[10] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*.
[11] W. Jia, K. A. Shaw, and M. Martonosi. 2014. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.
[12] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[13] Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[14] H. Kim, S. Hong, and H. Han. 2017. Compiler-Assisted Preloading in the Shared Memory for Thread-Dense Memory Requests. *Lecture Notes in Computer Science* 11027, 0 (2017).
[15] Y. Kim and A. Shrivastava. 2011. CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.
[16] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram. 2017. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*.
[17] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. 2014. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.
[18] S. Y. Lee and C. J. Wu. 2014. CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads. In *23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*.
[19] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[20] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and Transparent Cache Bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[21] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*.
[22] Chao Li, Yi Yang, Zhen Lin, and Huiyang Zhou. 2015. Automatic Data Placement into GPU On-Chip Memory Resources. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
[23] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. 2015. Priority-Based Cache Allocation in Throughput Processors. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.
[24] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU.
[25] NVIDIA. 2018. CUDA C Best Practice Guide.
[26] Reena Panda, Xinnian Zheng, Jiajun Wang, Andreas Gerstlauer, and Lizy K. John. 2017. Statistical Pattern Based Modeling of GPU Memory Access Streams. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*.
[27] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[28] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.
[29] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-Aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.
[30] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2014. Learning Your Limit: Managing Massively Multithreaded Caches Through Scheduling. *Commun. ACM* 57, 12 (Nov. 2014), 91–98.
[31] A. Sethia, D. A. Jamshidi, and S. Mahlke. 2015. Mascar: Speeding up GPU Warps by Reducing Memory Pitstops. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.
[32] Bin Wang, Zhuo Liu, Xinning Wang, and Weikuan Yu. 2015. Eliminating Intra-warp Conflict Misses in GPU. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*.
[33] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. 2015. DaCache: Memory Divergence-Aware GPU Cache Management. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*.
[34] B. Wang, Y. Zhu, and W. Yu. 2016. OAWS: Memory Occlusion Aware Warp Scheduling. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*.
[35] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*.
[36] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. 2015. Coordinated Static and Dynamic Cache Bypassing for GPUs. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.
[37] J. Zhang, S. Gao, N. S. Kim, and M. Jung. 2018. CIAO: Cache Interference-Aware Throughput-Oriented Architecture and Scheduling for GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
[38] Xian Zhu, Robert Wernsman, and Joseph Zambreno. 2018. Improving First Level Cache Efficiency for GPUs Using Dynamic Line Protection. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*.