Linebacker: Preserving Victim Cache Lines in Idle Register Files of GPUs

Yunho Oh*
EPFL
Lausanne, Switzerland
yunho.oh@epfl.ch

Gunjae Koo
Hongik University
Seoul, Korea
gunjae.koo@hongik.ac.kr

Murali Annavaram
University of Southern California
Los Angeles, USA
annavara@usc.edu

Won Woo Ro
Yonsei University
Seoul, Korea
wro@yonsei.ac.kr

ABSTRACT

Modern GPUs suffer from cache contention due to the limited cache size that is shared across tens of concurrently running warps. To increase the per-warp cache size prior techniques proposed warp throttling which limits the number of active warps. Warp throttling leaves several registers to be dynamically unused whenever a warp is throttled. Given the stringent cache size limitation in GPUs this work proposes a new cache management technique named Linebacker (LB) that improves GPU performance by utilizing idle register file space as victim cache space. Whenever a CTA becomes inactive, linebacker backs up the registers of the throttled CTA to the off-chip memory. Then, linebacker utilizes the corresponding register file space as victim cache space. If any load instruction finds data in the victim cache line, the data is directly copied to the destination register through a simple register-register move operation. To further improve the efficiency of victim cache linebacker allocates victim cache space only to a select few load instructions that exhibit high data locality. Through a careful design of victim cache indexing and management scheme linebacker provides 29.0% of speedup compared to the previously proposed warp throttling techniques.

KEYWORDS

GPU, cache, register file, CTA scheduling

ACM Reference Format:
Yunho Oh, Gunjae Koo, Murali Annavaram, and Won Woo Ro. 2019. Linebacker: Preserving Victim Cache Lines in Idle Register Files of GPUs. In *ISCA '19: International Symposium on Computer Architecture, June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322222

*This work was done while the author was at Yonsei University and USC.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322222

1 INTRODUCTION

Graphics Processing Units (GPUs) achieve high throughput by running thousands of threads concurrently. Threads are grouped into warps that each execute the same instruction in a single instruction multiple thread model. Multiple warps are packed into a cooperative thread array (CTA) which is assigned to a streaming multiprocessor (SM) in a GPU. Each GPU has tens of SMs, each concurrently executes tens of warps leading to massive parallel thread execution. This model benefits compute-intensive kernels that operate on small working sets. However, while executing even moderately memory-intensive workloads GPUs suffer significant slowdowns [3, 6, 14, 16, 25, 29, 42, 53, 55, 58, 63, 64]. One significant reason for the performance degradation in these workloads is the lack of sufficient cache space to store the working set of all the concurrently executing warps [30, 49]. The stringent cache space limitation continues to be a performance hurdle even though modern GPUs use multi-level caches and high-performance off-chip memory such as High-Bandwidth Memory (HBM) [1, 2]. Although recent GPU architectures have larger L1 caches than previous GPU generations [47, 48], a single thread still has access to just tens of bytes in L1 cache. Therefore GPUs suffer from severe cache thrashing. Even when there is strong data locality in a cache line (we use them term 'useful lines') it is frequently evicted before it is reused. Given the severity of cache limitation, even if a few of the useful lines are preserved for longer time significant speedups can be achieved in GPUs.

Several studies previously proposed cache bypassing [8, 37, 68] and warp throttling techniques [22, 38, 55] to improve the cache efficiency. Cache bypassing techniques reduce cache contention by selectively allocating the data fetched from the off-chip memory on L1 cache based on the expected locality of the fetched data. While bypassing improves the residency time of data that is already in the cache, it is a difficult tradeoff to decide the relative locality merits of incoming data and currently resident cache data. Even with the cache bypassing schemes, many cache lines are still evicted from L1 cache prior to their reuse. Warp throttling is an alternative technique that limits the number of active warps at runtime so as to increase the size of the cache space available per active warp. Warp throttling has a negative side effect of decreasing thread level parallelism. Hence, there is a tight tradeoff between the extent of throttling to improve per warp cache allocation and the detrimental effect of reduced parallelism.

GPUs rely on a fairly large register file to support fast thread switching across tens of warps. Each active warp stores its architectural context in its own dedicated register file space. The available register file space is one of the limiters of the number of warps allocated to an SM [50]. But when warp throttling is used to increase per warp cache space it has the unintended side effect of reducing the register file utilization. When an active warp is deactivated by a warp throttling scheme the register file space allocated to that warp is left unused until the warp is activated at a future time. Depending on the extent of warp throttling it is possible that tens of kilobytes of precious register file space is left idle due to reduced active warp count. Even without warp throttling register files in GPUs have been shown to already exhibit underutilization [15, 26, 31].

Considering the above challenges, in this paper we make a case for repurposing the idle register file space as an augmentation of cache space. We present a new GPU micro-architecture technique named Linebacker (LB) to enable idle register file space to be used as an extended L1 cache. Linebacker tracks the expected locality of cache lines evicted from L1 cache and uses the idle register file space as a victim cache to store these cache lines. Linebacker co-optimizes warp throttling and register file usage as a victim cache. Different from the previously proposed warp throttling techniques, linebacker's CTA throttling technique backs up the the registers of the warps of the CTAs that are throttled (i.e., inactive CTAs) to an off-chip memory space. Then, linebacker repurposes the register file space that is occupied by the registers of inactive CTAs as victim cache space. In the repurposed register file space, linebacker preserves only the useful lines after they are evicted from L1 cache. The selection of which data will be stored in victim cache relies on the observation that the static loads in GPUs exhibit the same cache behavior across warps [30, 49]. With this observation, linebacker identifies and separates the useful lines from other lines that can be simply dropped from cache. Linebacker considers a line that is accessed by a load exhibiting a certain level of data locality as a useful line. As a result, linebacker achieves significant performance improvement by a synergistic employment of the CTA throttling technique and the victim caching technique.

The algorithmic flow of linebacker is as follows. Linebacker determines the number of active CTAs by periodically monitoring the performance. When a CTA is throttled (and hence deactivated), linebacker backs up the registers used by the CTA to an off-chip memory space. The freed up register file space is tagged as available victim cache. When the deactivated CTA is reactivated at a future time the victim cache space is released and the register file space is restored back to the CTA.

For the victim cache management linebacker classifies the memory locality of each cache line by monitoring the program counter of the load that requested that cache line. If a data fetched by the load instruction is repeatedly used above a certain threshold then all cache lines requested by that load instruction are tagged as high locality cache lines. The basic assumption is that the locality behavior of the cache line is determined by the load that accessed that data. By monitoring the cache access patterns on a per load basis, linebacker categorizes a victim line as either useful or not at every cache miss. An useful victim line is then preserved in the register file victim cache. Because the size of a warp sized register is the same as a cache line size, linebacker performs a simple register-to-register move operation to effectively transfer a useful cache line that is being evicted into the victim cache space within a register file. Our evaluation shows that linebacker outperforms the previously proposed warp throttling technique [55] by 29.0%.

Table 1: Simulation configuration

# of SMs	16
Clock freq.	1126 MHz
SIMD width	32
Max. threads/warps/CTAs per SM	2048/64/32
Warp scheduling	Greedy-Then-Oldest (GTO), 4 schedulers per SM
Register file/SM	256 KB
Shared memory/SM	96 KB
L1 cache size/SM	16 / **48** / 64 / 96 / 128 KB 8-way, 128B line, 64 MSHRs
L2 shared cache	8-way, 2048 KB
Off-chip DRAM bandwidth	352.5 GB/s
Off-chip DRAM timing	RCD=12,RP=12,RC=40,RRD=5.5, CL=12,WR=12,RAS=28

Table 2: GPU applications for benchmark

App	Description	App	Description
S2	Symm. rank 2k operations [12]	GE	Scalar, Vector and Matrix Mul. [12]
BI	BiCGStab Linear Solver [12]	KM	KMeans [7]
AT	Matrix Transpose-Vector Mul. [12]	BC	BFS (CUDA SDK) [44]
S1	Symm. rank 1k operations [12]	MV	Matrix Vector Product-Transpose [12]
CF	CFD Solver [7]	PF	ParticleFilter Float [7]

(a) Cache-sensitive applications

BG	BFS (GPGPU-Sim) [5]	LI	LIBOR Monte Carlo [5]
SR2	SRAD (v2) [7]	SP	SPMV [60]
BR	BFS (Rodinia) [7]	FD	2D FDTD [12]
GA	Gaussian Elimination [7]	2D	2D Convolution [12]
SR1	SRAD (v1) [7]	HS	HotSpot [7]

(a) Cache-insensitive applications

2 MOTIVATION

In order to motivate the design of linebacker we provide some motivational data to show the viability of using register file as a victim cache and the algorithm used for determining the usefulness of a cache line. This motivational data is collected using our GPU microarchitecture simulator executing a wide range of kernels.

2.1 Simulation Configuration

We used GPGPU-Sim v3.2.2 in our experiments [5]. Table 1 describes the configurations of the baseline GPU architecture. We used a large collection of the applications from Parboil [60], Rodinia [7], Polybench [12], GPGPU-Sim [5] benchmark suites, and CUDA SDK [44] to collect motivational data. Table 2 list the benchmark applications we used. In case of applications that launch less than 5 CTAs by default, we increase the workload size to create sufficient number of CTAs so that all SMs have reasonable amount of work. Table 2 separates benchmarks into cache-sensitive and cache-insensitive categories. We classify applications that show more than 30% speedup with 192 KB cache compared to the baseline 48 KB cache as cache-sensitive applications.

2.2 Cache Inefficiency on GPUs

To analyze the severity of cache contention from multiple warps sharing a small cache in GPUs, we measured cold miss ratio and capacity/conflict miss ratio in each application. When a memory request causes an L1 cache miss while it accesses a line that was previously loaded, we consider that miss as a capacity miss. Since

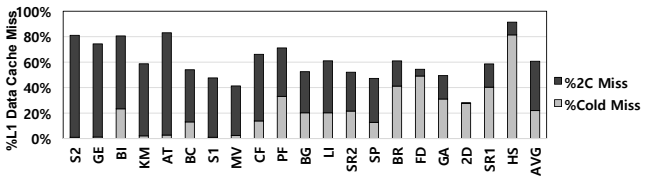

Figure 1: Breakdown of cold miss and capacity/conflict (2C) miss ratio (in our baseline)

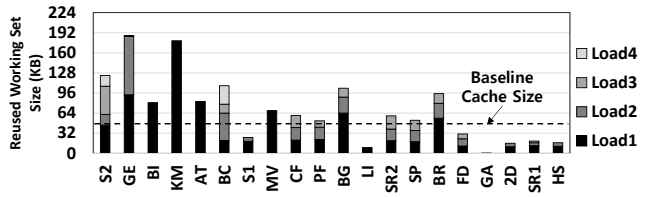

Figure 2: Per-SM working set size for top four frequently executed loads re-accessed within 50000 cycles period

capacity and conflicts misses are essentially a manifestation of the limited cache size [35, 37, 49], we consider them as a same category in our analysis.

Figure 1 shows the breakdown of cold miss ratio and capacity/conflict miss ratio of each application. In these results, the average L1 cache miss ratio (sum of cold miss ratio and capacity/conflict miss ratio) and the average capacity/conflict cache miss ratio are 66.6% and 44.6%, respectively. The capacity/conflict cache miss ratio accounts for 67.0% of total cache miss ratio. In 11 out of 20 applications, more than 70% of total L1 cache misses are classified as capacity/conflict misses. These results imply that GPUs frequently access L2 (or lower-level) cache or off-chip DRAM to fetch the data that is evicted from L1 cache again. Congestion of such long-latency memory operations increases stalls in the memory system. Due to these stalls, GPUs often suffer from performance degradation. If GPUs properly reuse cache lines before they are evicted, significant speedup can be achieved.

2.3 Behavioral Characteristics of GPU Loads

GPUs use single instruction multiple thread (SIMT) execution model where multiple warps execute the same instruction stream but on different data elements. The locality behavior of a load instruction in one warp tends to be the same as the locality exhibited by the same load instruction executed in a different warp. For example, if a load is processing streaming data in one warp then the same load is processing streaming data in other warps as well. Hence, such a load will exhibit poor temporal locality. Similarly, if a load in one warp hits in a cache the same load in another warp also tends to hit in the cache. This observation was also made in prior works [30, 49, 68].

Figure 2 shows the reused working set size of top four frequently accessed loads in each application. We consider a cache line to be reused if the line is re-accessed within a given time window (50000 cycles in our experiments). The total reused working set of just the top four loads (in KB) is plotted on the Y-axis for each benchmark. Note that we excluded any load that is processing streaming data when selecting the frequently accessed loads. Streaming loads are analyzed later.

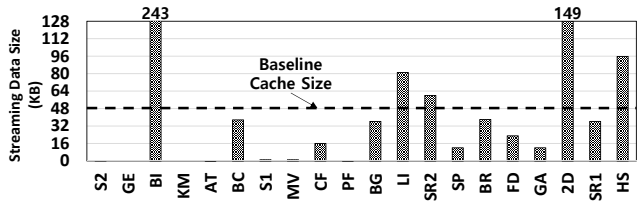

Figure 3: Per-SM streaming data size (50000 cycles period)

As shown in the figure, the aggregate working set size of four frequently accessed non-streaming loads exceeds the L1 cache size (48 KB) in 13 out of the 20 applications. Even though these loads some times share data with other loads, we observe that the majority of working set (85% in these workloads) is used solely by that load. As such any demand fetched data from one load is likely to be replaced by the data fetched by another load.

Some loads access streaming data. We consider a load to process streaming data if the cache miss ratio of only that load is more than 95% with an infinite cache in 50000 cycle window. We measure the size of data accessed by streaming loads. Figure 3 shows the experimental results. 9 out of 20 applications access more than 16 KB of streaming data, which corresponds to 33% of the L1 cache size. In BI, LI, SR2, 2D, and HS, the accessed streaming data exceeds the cache size. Apart from contention created by large working sets of frequently accessed non-streaming loads, streaming data further compounds the cache miss problem. Streaming data unnecessarily evicts reused cache lines.

2.4 Opportunity in GPU Register File

While cache is a highly contended resource, GPUs have a uniquely large register file. The large register file is often underutilized due to the following reasons. First, each SM in a GPU has a hardware limit on maximum number of warps that can be resident in the SM. If each of the warp uses a small number of registers then the total register space allocated for all warps resident in an SM may be less than the size of the register file [15, 26, 31, 70]. We call the register file space that is not occupied by any warp registers as *Statically Unused Register file (SUR)*. Second, several studies recently proposed warp throttling techniques that limit the number of active warps at runtime [22, 55]. By warp throttling, a portion of the warps may not be scheduled. Under this situation, the registers of inactive warps may not be accessed until they are scheduled again. We call the register file space occupied by the registers of inactive warps as *Dynamically Unused Register file (DUR)*. Statically unused register file space is known after compile time while dynamically unused register file space varies depending on the extent of warp throttling.

We measured the size of statically and dynamically unused register file space in our baseline GPU. Note that we compiled the applications with the default NVCC configuration of maximum register count (e.g., without setting *-maxrregcount* option). Warp throttling techniques dynamically change the number of active warps, so that the size of DUR may vary in each cycle. As an ideal case, we empirically determine the optimal number of warps for each application that provide the best performance, and assume optimistically that such a determination can be made statically. Such

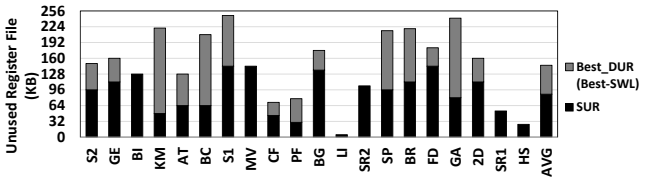

Figure 4: Size of Statically Unused Register file (SUR) and Dynamically Unused Register file (DUR) (with Static Warp Limit (SWL) configuration that achieve best performance for each application)

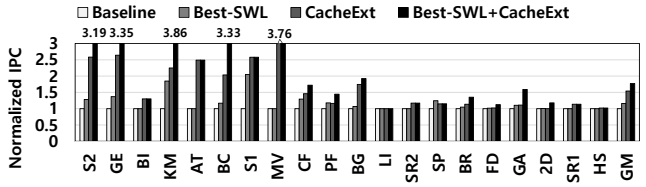

Figure 5: Performance effect of enhanced L1 cache (GM: Geometric Mean, All results are normalized to baseline)

an optimistic approach (we call Best-SWL) has been shown to provide better performance than dynamic warp throttling techniques such as Cache-Conscious Wavefront Scheduling (CCWS) [55]. In each application, we measure the size of dynamically unused register file space with the Best-SWL configuration by subtracting the size of registers used in the optimal number of warps from the size of registers used in the baseline.

Figure 4 shows the results of our investigation. The Y-axis shows the size of unused register space. The average size of statically unused register file space varies from 4 KB-144 KB with an average of 87.1 KB statically unused register space. In 13 out of 20 applications, Best-SWL leaves out 27 KB-173 KB (average 58.7 KB) of register space as dynamically unused.

Both the register file and the cache memory are essentially storage structures for transient data [4, 24, 50, 57]. We assume an idealized design that dynamically reassigns the unused register space as an extension of L1 cache. We measure the performance of GPUs with such an enhanced L1 cache as shown in Figure 5. The bars labeled *CacheExt* show the performance of the GPU that augments L1 cache with the size of statically unused registers in each application and just uses the baseline warp scheduling. The bars labled *Best-SWL+CacheExt* show the performance with Best-SWL scheduling and L1 cache that is dynamically augmented based on the size of statically and dynamically unused register file space. On average, Best-SWL shows 11.5% performance improvement compared to the baseline. On the other hand, CacheExt and Best-SWL+CacheExt show 54.3% and 77.0% of performance improvement, respectively. These results are 33.3% (CacheExt) and 53.0% (Best-SWL+CacheEx) better than Best-SWL. We find that warp throttling combined with large cache can show a synergetic effect on GPU performance.

Summary of Motivational Study: GPUs often suffer from performance degradation due to cache thrashing. A large portion of L1 cache lines that are re-accessed are from a small set of static load instructions. If GPUs preserve the cache lines fetched by those loads, significant speedups can be achieved. Previously, several studies proposed the warp throttling techniques to mitigate cache contention. While these techniques are effective, they cause the register file under-utilization at runtime.

Our goal is to design a new GPU microarchitecture that uses a new warp throttling schemes that works in conjunction with a cache augmentation scheme that reconfigures the underutilized register space as a cache.

3 LINEBACKER ALGORITHM

3.1 Key Concepts

Addressing the challenges and exploiting the opportunities we presented in the prior section we propose the LineBacker (LB) architecture. Linebacker is composed of three key concepts.

CTA Throttling: Linebacker uses a CTA throttling technique to find an optimal number of active CTAs while executing an application. Similar to the previously proposed warp throttling techniques, the proposed CTA throttling generally reduces active CTAs in cache-sensitive workloads. Linebacker determines the number of active CTAs by monitoring IPC performance periodically.

Inactive CTA Register Backup/Restore: The main difference between prior warp throttling techniques and linebacker is that it coordinates the throttling of warps with a register file backup and restore scheme. The register file space occupied by the registers of inactive CTAs becomes dynamically unused. Linebacker supports a register backup and restore scheme that works in conjunction with the warp throttling algorithm. If a CTA becomes inactive due to throttling, linebacker copies the registers associated the inactive CTA to an off-chip memory space. If an inactive CTA becomes active later linebacker restores the register files from memory as a high priority.

Selective Victim Caching Exploiting Idle Register File Space: As explained in Section 2.3, frequently reused data tend to be accessed by a small set of static loads. Linebacker exploits this property to select load instructions that exhibit strong locality to improve data reuse opportunities. To preserve the cache lines fetched by the selected loads, linebacker employs a victim caching scheme. Traditional victim caching schemes for CPUs reduce the conflict misses with a very few cache line entries [21, 59]. Without adding a large cache structure, linebacker utilizes idle register file space as a victim cache space. Linebacker is able to utilize not only statically unused registers but also dynamically unused registers as victim cache space. In some real-world GPU applications, the register file may be fully utilized at compile time [67]. In that case, statically unused register space may be very small or not existent. However, linebacker leans on CTA throttling, which is anyway necessary to reduce cache contention in cache sensitive applications, to create dynamically unused register space for victim caching.

The size of a thread operand value is 4-bytes in GPUs. Hence, a warp with 32 threads stores 32 × 4-bytes in each warp register. We match the size of the L1 cache to be the same as the size of the warp register (128 bytes) thereby making it easy to divert any evicted cache line to be stored in one warp register as a victim cache line.

3.2 Linebacker Workflow Example

Figure 6 shows a simple example that describes linebacker's workflow in an SM. This figure shows the execution flow of 6 periods (P0-P5). This illustration shows an SM with just 10 registers for

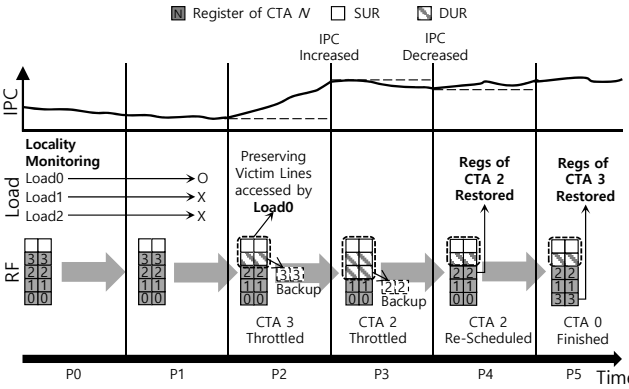

Figure 6: LB workflow example. 4 CTAs are launched by default. From the beginning to T0 is locality monitoring period. (RF: Register File)

simplicity. At the beginning of a kernel, each SM launches 4 CTAs and each CTA uses 2 registers. As such 2 out of 10 register file entries are statically unused.

Per-Load Locality Monitoring: Linebacker selects a subset of loads that exhibit high locality as candidates for victim caching. For this selection, linebacker counts the number of accesses to data fetched into L1 cache. As described earlier L1 cache data is frequently evicted before reuse and hence it is hard to accurately monitor data re-accesses only with L1 cache. To accurately monitor the per-load cache access behavior, linebacker uses Victim Tag Table (VTT). The victim tag table is originally designed to keep the tags of the victim lines that are preserved in the idle register file space. However, during the monitoring period, linebacker keeps only the tags of lines that are evicted from L1 cache and does not store the data of the victim lines.

When an SM executes a load, linebacker accesses the cache tag first to see if a requested memory address is present in the cache. On a cache miss linebacker accesses the victim tag table to see if that data was just evicted, in which case the victim tag table would have the tag stored. Linebacker counts the number of hits (in either L1 cache or victim tag table) and misses (in both L1 cache and victim tag table) within a time window. At the end of the window, if the total hits across cache and victim tags exceeds a predetermined threshold, the load is classified as high locality type.

To further improve confidence, linebacker observes the behavior for at least two consecutive time windows to see a consistent characterization of a load instruction. Only when a load is consistently characterized as high locality load for two consecutive time windows linebacker assigns victim cache space for the data associated with that load. In the example of Figure 6, we assume that the monitoring finishes in first two periods and linebacker selects only Load 0.

It is worth noting a few design choices in the locality monitoring algorithm. First, linebacker does not limit the number of loads that are tagged as high locality loads. Hence, if more than one load exhibits high locality during two consecutive time windows all the loads are tagged as high locality loads. Second, if more than one load exhibits high locality during a time window then the entire set of the same loads must also exhibit high locality in the next time window. If a subset of loads identified as high locality in the first window exhibit high locality in the second time window then linebacker does not tag any of the loads. Third, if no high locality loads are identified within the first two time windows then linebacker is disabled, with the assumption that the application is not cache sensitive. Finally, as long at least one load is identified as high locality in one time window monitoring continues into the next time window, until the high locality loads match across two time windows (or when the application completes execution).

CTA Throttling and Victim Caching: Once the load locality monitoring is completed and at least one load is identified as high locality in two consecutive time windows it is an indication that the application is cache sensitive. For cache sensitive applications linebacker starts with the assumption that warp throttling might help the application. As such, it starts throttling one CTA by deactivating the warps associated with that CTA. It then measures the fractional change in IPC between the current time window and previous time window. Linebacker calculates fractional IPC variation (IPC_Var(Prev, Cur)) as follows.

$$IPC\_Var(Prev, Cur) = \frac{IPC\_Cur - IPC\_Prev}{IPC\_Prev} \quad (1)$$

IPC_Cur and IPC_Prev are the IPC of the current period and that of the previous period, respectively. For example, the IPC variation between P0 and P1 (IPC_Var(P0, P1)) is (IPC_P1-IPC_P0)/IPC_P0.

In the example in Figure 6, the monitoring period ends after P1 and load 0 is identified as high locality. Hence, linebacker makes the assumption that warp throttling will benefit this application. It starts by throttling a CTA (CTA 3). It then copies the registers of the CTA to the off-chip memory. Then, linebacker invalidates those registers in the register file and utilizes the register file space occupied by the registers of CTA 3 and any of the statically unused registers as victim cache space. It then measures the IPC with the reduced active CTA count in the next time window. If the fractional IPC improves by more than a predetermined bound (10% in our experiments) linebacker throttles another CTA in the next time window. In the example figure at the end of P2, IPC_Var(P1, P2) is higher than 10%, so linebacker throttles one more CTA (CTA 2) and backs up its registers.

If too few warps run, GPUs may suffer from slowdown due to the underutilization of DRAM bandwidth or arithmetic cores [38]. Linebacker re-schedules an inactive CTA if it detects such slowdown. In this example, IPC_Var(P2, P3) either dropped or did not improve by more than 10% and hence linebacker re-schedules CTA 2 in P4 and restores the registers of CTA 2 to the register file.

Using the above described approach linebacker automatically tunes the warp throttling scheme for the best performance dynamically, and uses the registers assigned to throttled CTAs as victim cache space. Finally, when an active CTA (CTA 0) finishes in the middle of P5, a previously throttled CTA (CTA 3) is re-scheduled. If there is no inactive CTA, a new CTA is fetched for execution.

4 LINEBACKER ARCHITECTURE

Figure 7 describes the linebacker architecture. Shaded parts in this figure are newly added hardware to a GPU pipeline. The key modules are CTA Throttling Logic (CTL), Load Monitor (LM), and Victim

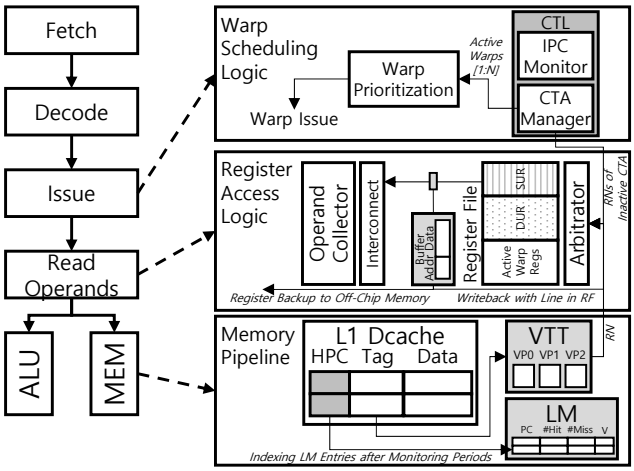

Figure 7: Linebacker architecture (RN: Register Number, HPC, Hashed PC, CTL: CTA Throttling Logic, VTT: Victim Tag Table, VP *N*: VTT Partition *N*, LM: Load Monitor)

Tag Table (VTT). Load monitor performs the monitoring functionality to identify high locality loads as described in the previous section. The load monitor relies on victim tag table structure to determine which load will benefit from a victim cache. CTA throttling logic determines appropriate level of CTA throttling and handles the register space saving and restoring process.

Load Monitor (LM): Load monitor logic counts hits (in L1 cache and the victim tag table) and misses made by each load within a given time window. Load monitor is a table consisting of the following fields: PC, hit, miss, and valid bit. The array is accessed with 5-bit hashed PC (HPC) of load. Rather than accessing the load minitor with the full PC of each load instruction, just a 5-bit hashed PC (XOR of 32-PC bits into 5 bits) was sufficient to preserve the per load PC behavior. Unlike CPU applications, GPU applications have very few global load instructions. In fact, the kernels in most applications we studied have fewer than 32 global loads. Hence, the hashed 5-bit PC was sufficient.

At the start of a monitoring time window, all the fields in each entry are initialized to zero. Every load that accesses the cache also concurrently accesses the LM table. If the load access is a hit in the cache then the hit count for the corresponding hashed PC entry in the LM table is incremented. If the load is a miss then the address is sent to L2 for fetching the data, but concurrently the address tag is searched in the victim tag table. If there is a hit in the victim tag table then the hit count is incremented in the LM table. The load miss also selects a victim cache line and the victim cache line tag is stored in the victim tag table. Note that the first access to LM table will also store the full PC of the load instruction that accesses the table.

At the end of a monitoring time window, all the LM entries whose hit ratios exceed a threshold will have their valid bit set to one. In essence all the entries whose valid bit is set to one are considered high locality loads since they either had more hits in the cache or in the victim tag table.

The above process is repeated in the second time window to build confidence in its determination of high locality loads. At the start of the second time window only the hit, miss counts are initialized to zero leaving the PC and valid bit information as is in each entry. The hit counters for each load are again incremented whenever there is a cache hit or a hit in the victim tag table, otherwise the miss count is incremented. At the end of the second time window any LM entry with a high hit ratio whose valid bit is already set to one is considered a high locality load. The monitoring period then stops.

To verify the load that lastly accesses a cache line, a hashed PC field is added to each cache line as shown Figure 7. This field is updated whenever the line is first fetched or accessed. If hashed PC of the victim line is same as the HPC of a selected load, linebacker verifies the victim line as useful.

Victim Tag Table (VTT): The victim tag table is a set of tag arrays that store the tags of any evicted cache line during the monitoring period. Once the monitoring period is completed it stores the tags of any victim cache lines that are stored in unused register space. The victim tag table has a set-associative tag array structures having the same number of sets as the L1 cache (48 sets). While the number of sets is the same as L1 cache the number of ways is determined by the amount of free register space. To modularize the design the VTT itself is partitioned and each VTT partition has 48 sets. In our preferred implementation linebacker requires a minimum of 24 KB unused register file space to be used a victim cache. With a 24 KB victim cache there are 192 victim cache lines of 128 bytes width. These 192 victim lines are divided into a 4-way set associative victim cache across the 48 sets. Thus a 4-way set associative victim tag array is used as a VTT partition and it is allocated for each 24 KB unused register space. The unused register file space is allocated at 24 KB granularity and any unused register space that is less than 24 KB is not used as a victim cache space. This approach simplifies the victim cache management and allocation policies.

Once the monitoring period is completed and candidate loads for victim cache storage are identified, then the victim tag table is used for accessing the victim cache space. When a memory request misses in L1 data cache, linebacker uses the same set index to access the victim tag table. Depending on the available register file space the number of ways in the victim tag table may vary. As described above the victim tag table is allocated in increments of four ways. While searching for a cache line that missed in L1 cache, linebacker searches the victim tag table in sequential order of the allocated partitions. On a hit in a VTT entry, linebacker generates the register file read request with the register number dedicated to the hit entry.

CTA Throttling Logic (CTL): CTL performs the following functions. First, CTL monitors IPC during each time window and determines the IPC change across two consecutive time windows. It uses an IPC monitor logic as shown in Figure 7. Recall that linebacker makes the assumption that warp throttling is beneficial and proactively throttles warps immediately after the end of the monitoring period. IPC monitor measures the effects of throttling warps by measuring the IPC variation across two consecutive time windows. The IPC variation threshold is used to either throttle a CTA or schedule an inactive CTA, as detailed earlier. The CTA throttling logic block includes a CTA manager that makes the throttling decisions. The CTA manager maintains a bit vector to track the scheduling status (active or inactive) of all CTAs assigned to an

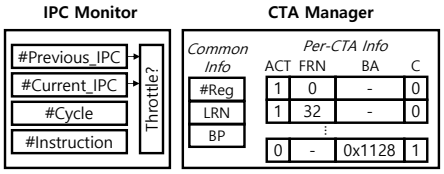

Figure 8: CTL Structure (BP: Backup Pointer, ACT: Active bit, FRN: First Register Number, BA: Backup Address, C: Backup Completed)

SM. CTA manager also performs back up or restore process of the architected register state associated with a CTA. The CTA manager keeps track of the physical register file indices assigned to a warp register [26] of active CTAs. The CTA manager also keeps track of the dedicated off-chip memory addresses where the registers of inactive CTAs are stored. When a CTA is throttled, the CTA manager finds the physical register file indices of the CTA and saves the register state to the off-chip memory dedicated for register storage. When a CTA is launched or re-activated, the CTA manager restores the registers with the architectural state data located at the previously stored memory address.

Delay Considerations: When a CTA is deactivated the register state associated with that CTA must be saved to memory before the register file space can be reused as a victim cache. Memory write latency can delay this process. To reduce this delay linebacker employs a 6-entry buffer where the registers are first written to and the buffer is drained asynchronously to the memory. Second, when a CTA is re-activated its register state must be restored from memory. Writing to register file may also be delayed to potential bank conflicts in the register file. To reduce the bank conflict related delays the same 6-entry buffer is used as a register accumulation buffer. Each buffer entry consists of an address field and a data field. The address field keeps the target register number or the off-chip memory address for backup. The data field keeps the content that is being saved or restored.

When a store instruction is executed, linebacker follows the default write policies of the baseline GPU, which are write-evict (hit) and write no-allocate (miss). When a store is executed, linebacker searches the requested line in both L1 data cache and the register file. If the written line hits in the register file, then that register entry is invalidated and the store data is sent directly to the next level of memory hierarchy. The invalidated register entry can be reused as a victim line later. When linebacker stores a new evicted line, one of the invalidated lines is replaced in priority. If a write miss occurs (in both data cache and register file), linebacker does not allocate the written line in either the primary or the victim cache. The net result of this store handling policy is that the data in the victim cache is never dirty and hence when a CTA is restored the data in the victim cache can be overwritten with the register file data.

4.1 Microarchitectural Details of Linebacker Implementation

CTL: Figure 8 depicts the CTA throttling logic structure. CTA throttling logic consists of an IPC monitor and a CTA manager. The IPC monitor has four elements that stores the IPC of the previous period (#Previous_IPC), the IPC of of the current period (#Current_IPC), the number of core clock cycles (#Cycle), and the number of completed instructions (#Instruction). At the start of a time window, #Cycle and #Instruction are reset to zero, and #Current_IPC is copied to #Previous_IPC. The monitor then counts clock cycles and instructions retired. At the end of a period, the IPC monitor calculates the IPC of current period by dividing the value in #Instruction by the length of a period. Then, the IPC monitor stores the IPC of current period in #Current_IPC and calculates the IPC_Var(Prev, Cur). After the initial monitoring periods, if the current kernel is classified as cache-sensitive and IPC_Var(Prev, Cur) is larger than the upper bound, the CTA throttling logic decreases the number of active CTAs by one. If IPC_Var(Prev, Cur) is smaller than the lower bound, CTA throttling logic increases the number of active CTAs by one.

CTA manager handles common information for all CTAs in a *Common Info* table. The common information includes the number of registers used by a CTA (#reg), the largest register number of active CTAs (LRN), and the memory address where linebacker can store the CTA registers when it is throttled. We refer to the memory address as Backup Pointer (BP). At the beginning of a kernel, CTA throttling logic updates #reg, BP, and LRN. The initial BP is a constant off-chip memory address. Also, CTA throttling logic sets the initial LRN to 0. Whenever IPC monitor throttles an active CTA, linebacker copies the registers of the CTA that is currently switched to inactive at the address stored in BP. After completing to back up the registers, CTA throttling logic updates LRN with the last RN of the currently switched CTA and BP by adding the previously stored BP to (#reg × 128). On the other hand, if an inactive CTA is re-scheduled, linebacker restores the registers backed up in the address (BP - #reg × 128) to the register file. After that, CTA throttling logic updates BP by subtracting #reg × 128 from the current BP value. Also, CTA throttling logic updates LRN with the largest RN of currently assigned registers.

CTA manager stores the information associated with each CTA in the *Per-CTA Info* table as shown in Figure 8. The *Per-CTA Info* is indexed with the hardware CTA IDs. As shown in Figure 8, a *Per-CTA Info* entry contains a CTA scheduling status bit (ACT) that indicates whether a CTA is active or not, a First Register Number (FRN) field that stores the register number of the first register of the CTA, and a Backup Address (BA) field that stores an off-chip memory address where the registers are backed up.

The *Per-CTA Info* entries are updated as follows. At the beginning of the kernel launch, the *Per-CTA Info* for each scheduled CTA is appropriately set. The ACT bit of each CTA that is scheduled for execution is set to true. FRN of each entry is updated with the physical register index number associated with R0 of the first warp in each CTA. If CTA throttling logic decides to reduce active CTAs, the ACT bit in the *Per-CTA Info* entry of the active CTA that has the largest hardware CTA ID is set to false. Then, CTA throttling logic sends the RN and the current BP to the temporal buffer shown in Figure 7 which then stores the registers in memory. After completing the register back up, CTA throttling logic flushes the FRN value in the corresponding entry and updates the BA with the BP value. Also, a bit that indicates the completion of register backup (C) is set to true.

Whenever there is an opportunity to schedule a CTA, CTA throttling logic searches an inactive CTA in priority. If an inactive CTA

exists, CTA throttling logic switches it to active status. Also, CTA throttling logic sets the ACT bit of the *Per-CTA Info* entry of the CTA to true and copies the registers as many as (#reg) backed up from the address stored in BA. After restoring the registers, CTA throttling logic updates FRN of the *Per-CTA Info* entry and the C bit in the CTA is set to false. If there is no inactive CTA, a new CTA is launched and the *Per-CTA Info* entry of currently completed CTA is updated.

LM: LM consists of a simple array that has 32 entries. As shown in Figure 7, each LM entry has three elements to monitor PC, the number of cache hits and misses caused by a load, and a 2-bit valid field. Each LM entry is accessed with 5-bit HPC. From beginning of a kernel, LM counts the number of cache hits and misses for each static load during the predetermined period (50000 core cycles). After the monitoring period, if a load hits in the cache/victim tag table above the predetermined threshold, LM categorizes that load as high locality load. We set the cache hit ratio threshold to 20% in our paper based on empirical observations.

In each period, LM sets the valid fields as follows. At the end of first period, LM shifts the value in the first bit of each valid field to the second bit. Then, LM updates the classification results of current period to the first bit. Then, linebacker compares two bits in each field. Only if two bits are set true, the corresponding loads are selected for victim cache storage.

VTT: As mentioned in Section 4, the victim tag table consists of multiple tag arrays implemented as VTT partitions (VP). Each tag array entry in VP stores a valid bit, a tag, and LRU bits. If a tag hits in an entry at set X and way Y of VP *N*, the register number associated with that victim cache line is calculated as follows.

$$RN = Offset + N_{VP} * \#\_VP\_entries + X * (\#\_ways) + Y \quad (2)$$

Offset in the above equation is the value that is manually set. In this paper, we set the Offset value to 511. Thus at most 1536 unused registers (RN 512-2047) can be mapped as victim line entries. To access the register file space, the read or write requests with the RNs which are not used as warp operands should be sent through the arbitrator. To support it, the victim tag table includes a simple logic that calculates the RNs as shown in Equation 2. Also, to store an evicted line, the write request with the same RN is generated.

At the beginning of a kernel or whenever an active CTA becomes inactive, the victim tag table compares the register number dedicated to the first entries in each VP to last register number in CTL. When a CTA becomes inactive, the victim tag table waits for the register backup to complete (by monitoring the C bit in the *Per-CTA Info* entry of the CTA). Once the backup is complete the victim tag table then activates the victim tag table partitions so that the freed up registers may be used for caching.

The victim tag table has 48 sets (corresponding to the 48 sets in L1 cache), and 32-way set-associative structure. Given 32 ways, a victim tag table partition can be 1-way to 32-way set-associative structure. Depending on the size of the idle register file space the set-associativity can be changed. 1-way associativity can utilize the idle register file space most efficiently, however searching for vicitm cache hits requires long latency to sequentially search multiple victim tag table partitions. On the other hand, a single 32-way set-associative victim tag table partition reduces the latency to search a tag, however it requires extremely large unused register file space.

Table 3: Microarchitectural configuration of Linebacker

IPC & Per-Load Locality Monitoring Period	50000 cycles		
Cache Hit Threshold	20%		
IPC Variation Bounds	Upper: 0.1 (10%), Lower: -0.1 (-10%)		
VTT Configuration	4-way set-associative VP / 8 VPs		
VP Access Latency	3 cycles		
CTA Manager Access Energy	1.94 pJ	**HPC Access Energy**	0.09 pJ
LM Access Energy	0.32 pJ	**VTT Access Energy**	2.05 pJ

We determine the best structure as 4-way set-associative, therefore up to 8 VPs can be used. The number of tag entries in a VP is 48 (sets)×4 (way)=192 in our design.

4.2 Overhead

For each L1 cache line, the size of HPC field is 5 bits. For 48 KB L1 cache, the total size of the field is 240 bytes. In LM, each entry uses a valid field (2 bits) and three 4-byte registers to store PC and monitor the cache hits and misses. LM consists of 32 entries, therefore it uses 392 bytes. IPC monitor has three 32-bit fields to store the two IPC values and the number of completed instructions. CTA manager has two 11-bit (#Reg and LRN), and a 32-bit (BP) storages. Also, *Per-CTA Info* has 32 entries. A *Per-CTA Info* entry consists of two 1-bit (ACT and C), an 11-bit (FRN), and a 32-bit registers. A tag entry in the victim tag table consists of 1-bit valid bit, 18-bit tag and 5-bit meta data. For 1536 victim tag table entries, total 4608 bytes are used. Each load buffer entry has a 32-bit address field and 128-byte storage for the line. Therefore, our 6-entry buffer has (4+128)×6=792 bytes. Overall linebacker requires 5.88 KB of storage overhead.

We analyze the space overhead of linebacker using CACTI [62]. We estimate that the area overhead is less than $0.19mm^2$. The estimated SM size based on GF100 die photo is $22mm^2$ [26, 31], the storage overhead of linebacker is 0.9% of an SM. Also, linebacker includes a handful logic structures such as simple comparators, multiplexers, an adder, and a multiplier. Those structures for arithmetic operations require negligible overhead compared to storage resources [31].

5 EVALUATION

The simulation configuration and the applications used in our evaluation are explained in Table 1 and Table 2, respectively. We simulated the microarchitecture details of linebacker, including modeling the monitoring period, victim tag table partition access latencies, register file bank conflicts, victim cache access details. The latencies and thresholds used in linebacker default implementation are shown in Table 3. If linebacker changes the number of active CTAs too frequently, the off-chip memory traffic overhead to transfer the registers may affect the performance. We experimentally configured the IPC variation bounds that prevent frequent throttling and re-activating CTAs. Also, we modeled the GPU power consumption including backing up and restoring registers between the register file and the off-chip memory with GPUWattch [34]. We modeled the energy consumption of the CTA manager, HPC fields, LM, and the victim tag table with CACTI [62].

We compare linebacker to the following three approaches. We use Best-SWL as an idealized warp throttling scheme. As described

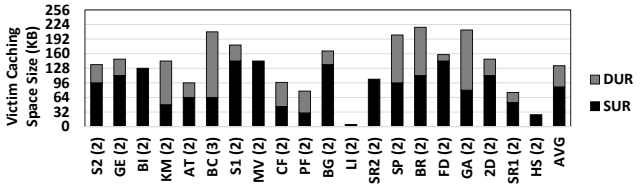

Figure 9: Size of idle register file space (victim caching space) and per-locality monitoring periods in Linebacker (Numbers in parentheses are the number of periods that Linebacker spends to find the loads having high data locality.)

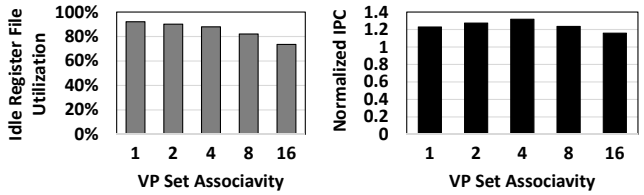

Figure 10: Idle register file utilization (left) and performance (right) according to set associativity of a single VTT Partition (VP) (Performance results are normalized to Best-SWL.)

earlier, this approach uses an oracle to determine the optimal number of CTAs to schedule during each time window but does not rely on victim caching. We also implemented a combination of dynamic warp throttling and cache bypassing introduced in Priority-based Cache Allocation (PCAL) scheme [38], and an unified on-chip local memory architecture [18]. PCAL determines how many cache lines are allowed for a given warp based on monitoring performance variations and other microarchitectural events in the GPU memory system. The warps that have no authority to allocate data in L1 cache simply bypass L1. The unified on-chip local memory structure which we implemented was proposed in Cache-Emulated Register File (CERF) scheme [18]. Among two previously proposed unified on-chip local memory structures [11, 18], we observe that CERF outperforms on-chip local memory structure proposed in [11] by utilizing the rarely reused register file space as the cache space. The size of CERF in the experiments is 304 KB, the aggregated size of the register file (256 KB) and L1 cache (48 KB). With the same baseline used in this paper, we estimate the hardware overhead of CERF is almost same as linebacker (CERF: 5.64 KB, LB: 5.88 KB).

Figure 9 shows the size of static and dynamic unused register file space that can be used as victim cache space by linebacker. The size of dynamically unused register space may vary in each time window. We calculate the average dynamic size by dividing the sum of dynamic unused space in each cycle by total execution cycles. The average size of dynamic and static unused spaces are 48.5 KB and 88.5 KB, respectively. Also, as shown in this figure, we observed that linebacker finds the loads having high locality within two periods in most applications.

5.1 Effect of Victim Tag Table Partition Set Associativity

We analyze the effect of set associativity of victim tag table partitions. As mentioned in Section 4.1, the size of available victim cache space varies and accordingly the available set associativity of victim tag table also varies. Requiring higher associativity in victim tag table partitions reduces the opportunity for victim caching since it demands larger unused register space. To analyze such trade off, Figure 10 shows the utilization of unused register file and the performance of each victim tag table configuration. Among the configurations, linebacker shows the best performance when using 4-way set associative victim tag table partitions (29.0% performance improvement). With 4-way set-associative design 88.5% of unused register file is used as victim line storage. As such linebacker employs 4-way set-associative victim tag table partitions in its default design.

When using a 1-way victim tag table partition configuration, it utilizes 92.8% of unused register file space. However, the performance results are not as high due to long victim tag table searching latency. On the other hand, using 16-way set-associative victim tag table partitions leads to many missed opportunities to reuse registers as victim cache, and on average only 71.1% of unused register file space is used as victim cache.

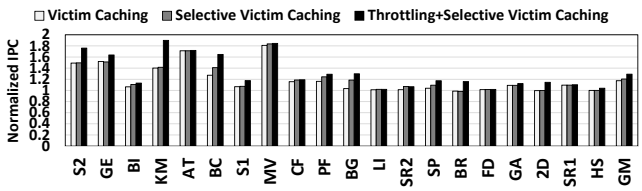

Figure 11: Performance comparison according to techniques in Linebacker (GM: Geometric Mean, All results are normalized to Best-SWL.)

5.2 Linebacker Performance Breakdown

We analyze the performance effect of various enhancements proposed within Linebacker. Figure 11 shows the experimental results. In this figure, *Victim Caching* is the configuration that preserves all evicted lines without doing any monitoring of high locality loads. As a result in this configuration even streaming data may be stored in victim cache. With *Selective Victim Caching*, linebacker selects the loads having high locality first and preserves the victim lines accessed by these selected loads only in the statically unused register space. Since there is no CTA throttling there is no dynamically unused register space available in this configuration. *Throttling+Selective Victim Caching* includes all techniques that we designed.

Selective Victim Caching achieves more than 7% of performance improvement compared to *Victim Caching* in BI, BC, BG, SR2, and SP. These applications access large-sized streaming data as shown in Figure 3. With these results, we observed that *Selective Victim Caching* efficiently reduces a large amount of cache contentions from streaming data. *Throttling+Selective Victim Caching* achieves 7.7% performance improvement compared to *Selective Victim Caching*. CTA throttling increases available register file size which enables the active CTAs to use more register space. Hence this combination shows a synergetic effect of using enough CTAs to exploit parallelism while simultaneously curtailing unnecessary CTAs to maximize victim cache space.

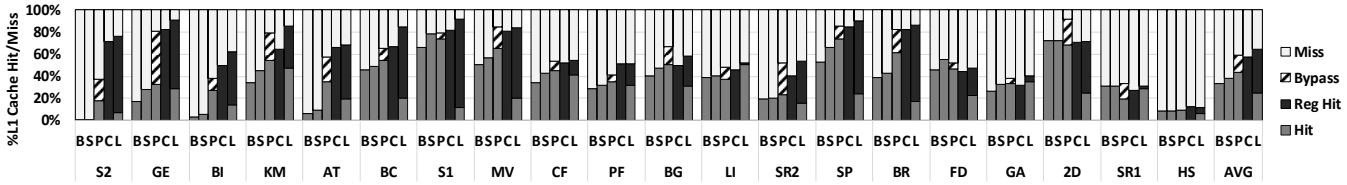

Figure 13: L1 cache hit/miss, victim cache hit in Linebacker (Reg Hit), and bypass breakdown (B: Baseline, S: Best-SWL, P: PCAL, C:CERF, L: Linebacker)

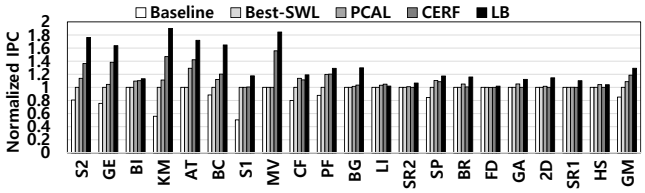

Figure 12: Performance comparison between Linebacker and other approaches (Normalized to Best-SWL)

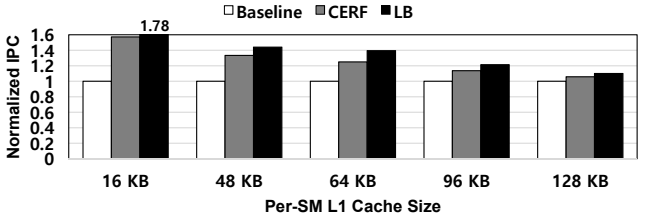

Figure 14: Performance comparison varying L1 cache size (All results are geometric mean values for applications used in this paper. The results are normalized to baseline with each cache configuration.)

5.3 Comparison with Previous Approaches

Performance (IPC): Figure 12 shows the performance of the baseline (listed in Table 1), Best-SWL, PCAL, CERF, and linebacker. Linebacker achieves 29.0% performance improvement over Best-SWL, which is the best among all architectures. PCAL and CERF improve performance by 7.6% and 19.6% compared to the Best-SWL, respectively. As a result, linebacker achieves more performance improvement than the baseline, PCAL, which is the combination of warp throttling and cache bypassing, and CERF that utilizes unused register file as L1 cache, by 44.1%, 24.0%, and 9.4%, respectively.

Both linebacker and CERF provision a large-sized cache, therefore they can achieve more speedup than PCAL as analyzed in Section 2.4. In S2, GE, KM, BC, BG, SP, and BR, linebacker utilizes a large dynamically unused register space as victim cache space as shown in Figure 9. Also, linebacker properly filters the victim lines from streaming data in BI, BC, BG, and BR as observed in Section 5.2. Due to these reasons, linebacker outperforms CERF in those applications. In FD and 2D, which are relatively cache-insensitive applications, CERF incurs numerous bank conflicts between cache accesses and register accesses so that latency of both register and cache accesses increase. So, CERF achieves lower performance than linebacker.

Cache Hit and Miss Ratio: To further analyze how linebacker improves the performance, we analyze memory requests. Figure 13 shows the breakdown of cache hit/miss and bypass. Each request results in either L1 cache hit (Hit), miss (Miss), L2 cache or off-chip memory access with L1 cache bypass (Bypass), or Reg hit (cache hit in register file, for CERF and linebacker). Latency of a Reg hit is slightly longer than an L1 cache hit due to multiple victim tag table accesses, queuing delay in the arbitrator and the bank conflicts in the register file. However, this latency is still much shorter than the latency of L2 cache access (minimum 200 cycles) or off-chip memory access. In case of Bypass access, the latency is similar to the L1 cache miss penalty. However, these accesses reduce the cache contention, so they contribute to performance improvement.

Linebacker achieves the best cache hit ratio among all architectures. The aggregated Reg hit and cache hit ratio of linebacker is 65.1%. On average, 40.4% of total memory accesses result in Reg hits. In linebacker, a large portion of victim lines are not fetched in L1 cache again while warps re-access them. Due to this reason, linebacker shows smaller L1 cache only hit ratio than the baseline. Instead, it achieves high Reg hit ratio (which is a combination of L1 and victim cache hits). The cache hit (Reg hit) ratio of CERF 57.9%. In BI, BC, BG, and BR, linebacker encounters more cache hits than CERF by filtering streaming data. Also, linebacker achieves higher cache hit ratio in KM, BC, BR by enhancing victim cache space using dynamically available registers due to CTA throttling.

5.4 L1 Cache Size Impact on Linebacker

L1 cache size varies according to GPU architectures. For example, in the NVIDIA Volta architecture, each SM can utilize up to 128 KB L1 cache [47] while the Maxwell and Pascal architectures include 48 KB L1 cache per SM [43, 45, 46]. To see the effect of linebacker on various cache configurations, we measured the performance of linebacker with different L1 cache sizes in this section. For comparison, we also measured the performance of CERF. In each cache configuration, the size of unified on-chip local memory of CERF is the sum of the register file size (256 KB) and L1 cache size.

Figure 14 shows the experimental results. For all configurations, linebacker achieves better performance than CERF. With the smallest cache configuration (16 KB), linebacker shows 78.0% performance improvement while CERF shows 58.1%. With the largest cache (128 KB), linebacker still outperforms CERF. Linebacker achieves 12.0% performance improvement while CERF shows 6.1% performance improvement in this configuration. Even with a large cache space, warps still access working set which is larger than the L1 cache size in some applications. As mentioned in Section 5.2, linebacker further enhances the effective L1 cache space with victim cache, so it could achieve better performance gain than CERF. Also, in the applications that access a large-sized streaming data, such as BI and BC shown in Section 2.3, CERF could not completely resolve cache thrashing caused by the large streaming data even

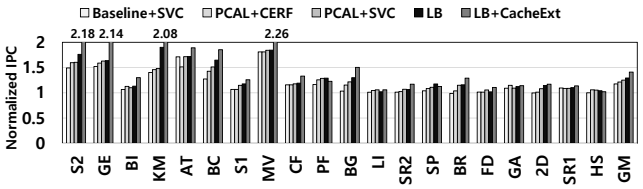

Figure 15: Performance comparison between combinations of warp scheduling and cache memory structures (GM: Geometric Mean, SVC: Selective Victim Caching, All results are normalized to Best-SWL.)

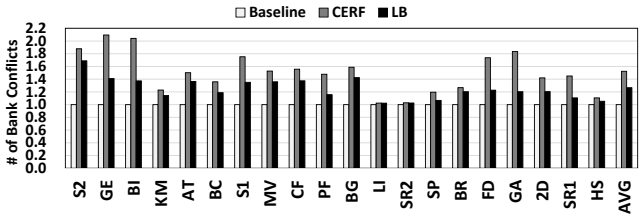

Figure 16: Comparison of number of bank conflicts in register file (Normalized to Baseline)

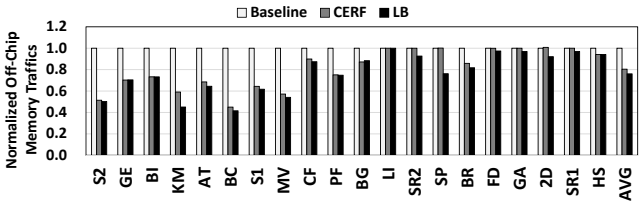

Figure 17: Memory traffic comparison (Normalized to Baseline. Linebacker overhead is additional memory traffics used for register backup and restore. In all applications, linebacker overhead is less than 1% of overall memory traffic.)

with an enhanced cache space. On the other hand, linebacker properly filters the streaming data, so it achieves better performance than CERF. Thus linebacker can be an efficient solution to improve cache efficiency for a wide range of GPU architectures.

5.5 Various Combinations of Previous Works

To further analyze the performance effect of linebacker, we compare linebacker to three combinations of the previous warp scheduling and cache architectures. First, we consider the combination of *PCAL+CERF* which can show the performance benefit of PCAL with a large L1 cache. Also, to evaluate the synergistic effect of the proposed CTA throttling and Selective Victim Caching (SVC) of linebacker, we consider two different variants: the baseline GPU with SVC (*Baseline+SVC*) and PCAL with SVC (*PCAL+SVC*). These architectures utilize only statically unused register space as victim cache space. *PCAL+SVC* supports warp throttling, however it does not utilize dynamically unused registers as victim cache space. Finally we measured the performance of linebacker with CacheExt, which is a well-balanced register file and L1 cache configuration as mentioned in Section 2.4.

Figure 15 shows the experimental results. linebacker outperforms Baseline+SVC, PCAL+CERF, and PCAL+SVC. Note that *Baseline+SVC* is identical to the *Victim Caching* configuration in Section 5.2. *PCAL+CERF* achieves 21.3% performance improvement and it is slightly (1.7%) better than CERF only. In this architecture, CERF provisions a large-sized L1 cache, however a portion of streaming data that cannot be filtered by PCAL still causes a large number of early evictions. Therefore, *PCAL+CERF* could not improve the performance further. *PCAL+SVC* achieves 25.1% performance improvement by filtering streaming data, however it is still 4.1% lower than linebacker. With these results, we find that it is more beneficial to utilize the SUR space as victim cache space than to keep the registers of the warps in cache-sensitive workloads.

LB+CacheExt achieves 41.9% performance improvement, while the Baseline with CacheExt shows 33.3% improvement as shown in Section 2.4. With these results, we observe that selective victim caching and utilizing DUR as victim cache space in linebacker can contribute to performance improvement even with an ideal register file and cache configuration.

5.6 Bank Conflicts in Register File

Frequent bank conflicts increase the latency to access the warp registers. We compare the number of bank conflicts of linebacker to that of CERF, which shows the best performance among the previous approaches in this section. Figure 16 shows the experimental results. All results are normalized to the baseline. Both CERF and linebacker increase the bank conflicts compared to the baseline by 52.4% and 29.1%, respectively. Using registers as victim cache lines increases the total number of register file accesses, therefore the bank conflicts increase.

Linebacker however has fewer bank conflicts than CERF due to the following reasons. While CERF stores all cache lines and the registers in a unified memory space, linebacker excludes the cache lines from streaming data to keep in the register file. Eliminating streaming data reduces the number of writes in the register file. Linebacker has more L1 cache hits. When a hit is in L1 cache, linebacker does not access the register file to fetch a cache line. Therefore, linebacker achieves less bank conflicts than CERF.

5.7 Off-Chip Memory Traffic

We compare the off-chip memory traffic of linebacker to those of CERF. Figure 17 shows the comparison results. Linebacker reduces the off-chip memory traffic by 24.0% compared to the baseline. Compared to CERF, linebacker reduces 4.6% more memory traffic. Also, we observe that the overhead of register backup and restore in linebacker is negligible. While the size of dataset used in each application is hundreds or thousands of megabytes, overall amount of traffic utilized for register backup and restore is only hundreds of kilobytes or a few megabytes as linebacker prevents unnecessary activation and deactivation of CTAs using a 10% IPC variation threshold.

5.8 Power and Energy Consumption

Linebacker utilizes the unused register file space, therefore they consume more power than the baseline GPU. The newly designed microarchitecture modules in linebacker cumulatively use less than 5% of power consumed by the register file. The additional power consumption at the GPU chip level is only 0.75% [13, 34, 40]. Also,

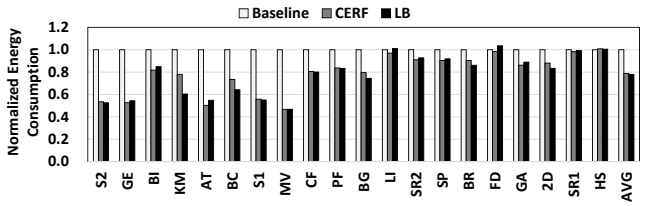

Figure 18: Energy consumption comparison (Normalized to Baseline)

the additional overhead of data traffic for register backup/restore is negligible as explained in the previous section. So, the power overhead in GPU memory system in linebacker is quite low.

We compared the energy consumption of linebacker, CERF, and baseline. Figure 18 shows the experimental results. Linebacker reduces the energy consumption by 22.1% on average compared to the baseline. Linebacker significantly reduces the execution time while consuming small amount of additional power. CERF reduces 21.2% of energy consumption compared to the baseline. Linebacker effectively improves the energy efficiency as well as GPU performance.

Considering the experimental results in this section and the hardware cost estimation, we believe that linebacker is a cost-effective solution to exploit the data locality on GPUs.

6 RELATED WORK

Warp/CTA Scheduling and Throttling: Several warp scheduling techniques were introduced to improve the cache efficiency or the performance of memory-intensive workloads [9, 19, 20, 32, 33, 49, 58]. Also, there are several studies that proposed CTA scheduling techniques for concurrent kernel executions on GPUs [51, 52, 61, 69].

Warp throttling techniques particularly focused on mitigating cache contention on GPUs. Rogers et al. proposed the Cache Conscious Wavefront Scheduling (CCWS) [55]. It determines the number of active warps based on the locality scoring at runtime. They also proposed another warp throttling technique, which considers memory divergence along with locality preservation [56]. Mao et al. introduced a warp throttling technique, which mathematically estimates the effective working set size and correspondingly an appropriate number of active warps [41]. Wang et al. proposed a new warp throttling technique that regulates active warps while a GPU runs multiple programs concurrently by monitoring effective bandwidth [66]. Kayiran et al. introduced a warp throttling technique at CTA granularity to improve the resource utilization including memory resources [22]. Wang et al. proposed a warp throttling technique based on the Miss Status Holding Register (MSHR) status [65].

Cache Bypassing: Jia et al. proposed a compiler-based technique to selectively allocate the cache lines on GPUs [16]. They also proposed a cache management scheme combining memory request prioritization and cache bypassing [17]. Li et al. proposed a cache bypassing technique based on locality monitoring [37]. Xie et al. introduced a coordinated static and dynamic cache bypassing technique [68]. Also, there are selective cache bypassing techniques according to the specific static loads [8, 36]. Li et al. proposed a orchestrated warp throttling and cache bypassing to improve the GPU performance [38].

Cache Structure and Management: Several studies introduced new cache management techniques or new cache memory structures for GPUs. Wang et al. proposed a GPU cache management technique which allocates the cache lines considering the warp scheduling priorities [64]. Komuravelli et al. introduced a new on-chip memory structure, called Stash [28]. Stash exploits the advantages of cache (globally addressable) and scratchpad memory (directly addressed). Gebhart et al. introduced a unified and reconfigurable multi-banked memory for cache, registers, and scratchpad memory according to the application features. [11]. To improve the cache efficiency of unified register file space, Jing et al. proposed the cache-emulated register file architecture [18]. This architecture uses the space for rarely accessed register data as the data cache, therefore the effective cache size can be enhanced than the unified memory structure proposed by Gebhart et al. [11]. Rhu et al. proposed a new L1 cache structure whose cache line size is configurable according to cache access characteristics [54]. Li et al. introduced tag-split cache structure to improve the cache utilization [39].

Victim Cache: Victim cache architecture was originally proposed for CPUs. Jouppi firstly proposed the concepts of miss cache and victim cache [21]. After then, several studies proposed the improved concepts of victim cache for various processor architectures [21, 23, 59, 71]. Recently, Gaur et al. proposed a new cache compression technique to improve the efficiency of victim caching [10].

Register File Utilization: In GPUs, the register file is often underutilized. Addressing this problem, several studies proposed new techniques to efficiently utilize the register file. Jeon et al. proposed a new technique that keeps only live registers in the register file and accesses them using a register renaming technique [15]. Kloosterman et al proposed another technique that reduces the power consumption of register file by employing a small-sized hardware buffer instead of the register file and properly preloads live registers to that buffer [27]. Xie et al. proposed a new compiler technique that adaptively determines the number of per-thread registers with the compile-time workload characteristic analysis [67].

Different from the previous studies, Linebacker aims to increase the effective cache size by re-purposing idle register file space. Exploiting load characteristics on GPUs, the proposed Linebacker architecture efficiently manage the lines evicted from cache and make warps access quickly.

7 CONCLUSION

L1 cache lines in GPUs are frequently evicted but are also re-referenced later. Most re-referenced evictions are from a small set of loads exhibiting high memory locality. Based on this observation this work proposes the Linebacker architecture. Linebacker dynamically identifies the evicted lines that are likely to be reused in future. Once this identification is done Linebacker then repurposes unused registers from GPU to store the evicted cache lines. Thus the register file acts as a victim line storage for re-referenced cache lines. To prevent cache pollution Linebacker also detects streaming loads and eliminates their accesses from victim cache storage. Linebacker

provides CTA throttling and the backup/restore of registers of inactive CTAs. CTA throttling increase the available victim cache space as well as reduce cache contention. Thus linebacker selects an appropriate number of CTAs to fully exploit workload parallelism, while allowing unused register files to be used as enhanced cache space. Our simulation results show that Linebacker achieves 29.0% of performance improvement compared to state of the art warp throttling techniques.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2018R1A2A2A05018941), by Institute of Information & Communication Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation), and by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053, NSF grants 1719074. W. W. Ro is the corresponding author.

REFERENCES

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[2] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.

[3] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

[4] Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael Abu-Ghazaleh. 2019. CORF: Coalescing Operand Register File for GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 701–714. https://doi.org/10.1145/3297858.3304026

[5] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*.

[6] N. Chatterjee, M. O'Connor, G.H. Loh, N. Jayasena, and R. Balasubramonia. 2014. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*.

[7] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009. IEEE International Symposium on Workload Characterization (IISWC '09)*.

[8] Xuhao Chen, Li-Wen Chang, C.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-47)*.

[9] Ahmed ElTantawy and Tor M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *Proceedings of the 2018 IEEE 24th International Symposium on High Performance Computer Architecture (HPCA '18)*.

[10] J. Gaur, A. R. Alameldeen, and S. Subramoney. 2016. Base-Victim Compression: An Opportunistic Cache Compression Architecture. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*.

[11] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*.

[13] Sunpyo Hong and Hyesoon Kim. 2010. An Integrated GPU Power and Performance Model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*.

[14] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*.

[15] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU Register File Virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[16] 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 '12)*.

[17] W. Jia, K. A. Shaw, and M. Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*.

[18] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, and X. Liang. 2016. Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*.

[19] 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)*.

[20] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*.

[21] Norman P. Jouppi. 1990. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*.

[22] O. Kayiran, A. Jog, M.T. Kandemir, and C.R. 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 '13)*.

[23] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. 2010. Using Dead Blocks As a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*.

[24] F. Khorasani, H. Asghari Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar. 2018. RegMutex: Inter-Warp GPU Register Time-Sharing. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[25] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*.

[26] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram. 2016. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *Proceedings of the 2016 IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA '16)*.

[27] John Kloosterman, Jonathan Beaumont, D. Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. 2017. Regless: Just-in-time Operand Staging for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

[28] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[29] G. Koo, H. Jeon, and M. Annavaram. 2015. Revealing Critical Loads and Hidden Data Locality in GPGPU Applications. In *2015 IEEE International Symposium on Workload Characterization (IISWC '15)*.

[30] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*.

[31] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-compression: Enabling Power Efficient GPUs Through Register Compression. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[32] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. 2015. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[33] Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*.

[34] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling

Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*.

[35] 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 '17)*. ACM, 297–311.

[36] 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 '15)*.

[37] 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 '15)*.

[38] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Keckler. 2015. Priority-based cache allocation in throughput processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*.

[39] Lingda Li, Ari B. Hayes, Shuaiwen Leon Song, and Eddy Z. Zhang. 2016. Tag-Split Cache for Efficient GPGPU Cache Utilization. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*.

[40] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. 2014. Power Modeling for GPU Architectures Using McPAT. *ACM Trans. Des. Autom. Electron. Syst.* 19, 3 (June 2014).

[41] Mengjie Mao, Jingtong Hu, Yiran Chen, and Hai Li. 2015. VWS: A versatile warp scheduler for exploring diverse cache localities of GPGPU applications. In *Design Automation Conference, 2015 52nd ACM/EDAC/IEEE (DAC '15)*.

[42] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

[43] NVIDIA. 2014. *NVIDIA GeForce GTX 980: Featuring Maxwell, The Most Advanced GPU Ever Made.*

[44] NVIDIA. 2016. *NVIDIA CUDA SDK Code Sample 4.0.*

[45] NVIDIA. 2016. *NVIDIA GeForce GTX 1080: Gaming Perfected.*

[46] NVIDIA. 2016. *NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built.*

[47] NVIDIA. 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE.*

[48] NVIDIA. 2018. *NVIDIA Turing GPU Architecture: Graphics reinvented.*

[49] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. 2016. APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*.

[50] Yunho Oh, Myung Kuk Yoon, William J. Song, and Won Woo Ro. 2018. FineReg: Fine-Grained Register File Management for Augmenting GPU Throughput. In *2018 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*.

[51] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2014. Preemptive Thread Block Scheduling with Online Structural Runtime Prediction for Concurrent GPGPU Kernels. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*.

[52] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. 593–606.

[53] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2016. A case for toggle-aware compression for GPU systems. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*.

[54] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.

[55] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[56] 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)*.

[57] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. 2018. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*.

[58] A. Sethia, D. A. Jamshidi, and S. Mahlke. 2015. Mascar: Speeding up GPU warps by reducing memory pitstops. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*.

[59] D. Stiliadis and A. Varma. 1997. Selective victim caching: a method to improve the performance of direct-mapped caches. *IEEE Trans. Comput.* 46, 5 (1997).

[60] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

[61] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 193–204. https://doi.org/10.1109/ISCA.2014.6853208

[62] Shyamkumar Thoziyoor, Naveen Muralimanohar, and Jung Ho Ahn. 2008. *Cacti 5.1.* Technical Report. Hewlett-Packard Laboratories.

[63] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[64] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. 2015. DaCache: Memory Divergence-Aware GPU Cache Management. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS '15)*.

[65] Bin Wang, Yue Zhu, and Weikuan Yu. 2016. OAWS: Memory Occlusion Aware Warp Scheduling. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques (PACT '16)*.

[66] Haonan Wang, Fan Leon Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *Proceedings of the 2018 IEEE 24th International Symposium on High Performance Computer Architecture (HPCA '18)*.

[67] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[68] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*.

[69] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 230–242. https://doi.org/10.1109/ISCA.2016.29

[70] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. 2016. Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*.

[71] Michael Zhang and Krste Asanovic. 2005. Victim Replication: Maximizing Capacity While Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*.