Priority-Based Cache Allocation in Throughput Processors

Dong Li[§†], Minsoo Rhu[§†‡], Daniel R. Johnson[‡], Mike O'Connor[†‡],
Mattan Erez[†], Doug Burger[*], Donald S. Fussell[†] and Stephen W. Keckler[†‡]
[†]The University of Texas at Austin, [‡]NVIDIA, [*]Microsoft

***Abstract*—GPUs employ massive multithreading and fast context switching to provide high throughput and hide memory latency. Multithreading can increase contention for various system resources, however, that may result in suboptimal utilization of shared resources. Previous research has proposed variants of throttling thread-level parallelism to reduce cache contention and improve performance. Throttling approaches can, however, lead to under-utilizing thread contexts, on-chip interconnect, and off-chip memory bandwidth. This paper proposes to tightly couple the thread scheduling mechanism with the cache management algorithms such that GPU cache pollution is minimized while off-chip memory throughput is enhanced. We propose priority-based cache allocation (PCAL) that provides preferential cache capacity to a subset of high-priority threads while simultaneously allowing lower priority threads to execute without contending for the cache. By tuning thread-level parallelism while both optimizing caching efficiency as well as other shared resource usage, PCAL builds upon previous thread throttling approaches, improving overall performance by an average 17% with maximum 51%.**

I. Introduction

GPUs have become the dominant co-processor architecture for accelerating highly parallel applications as they offer greater instruction throughput and memory bandwidth than conventional CPUs. Such chips are being used to accelerate desktops, workstations, and supercomputers; GPU computing is also emerging as an important factor for mobile computing.

GPUs rely on massive multithreading to tolerate memory latency and deliver high throughput. For example, a modern NVIDIA Kepler GPU can have up to 30,720 threads resident in hardware and operating simultaneously. GPU memory systems have grown to include a multi-level cache hierarchy with both hardware and software controlled cache structures. The high-end NVIDIA Kepler now includes a total of nearly a megabyte of primary cache (including both hardware and software controlled) and 1.5MB of L2 cache. Exploiting locality in throughput processors will increase in importance to reduce both power and off-chip bandwidth requirements. However, with so many threads sharing the caches and other shared resources, the overall system often suffers from significant contention at various levels. In particular, for applications that are performance-sensitive to caching efficiency, such contention may degrade the effectiveness of caches in exploiting locality, thereby suffering from significant performance drop inside a "performance valley" (PV region in Figure 1) – an observation made by Guz et al. that performance becomes very low at a *middle ground* between mostly cache-hits (MC region) and mostly cache-misses (MT region) [1].

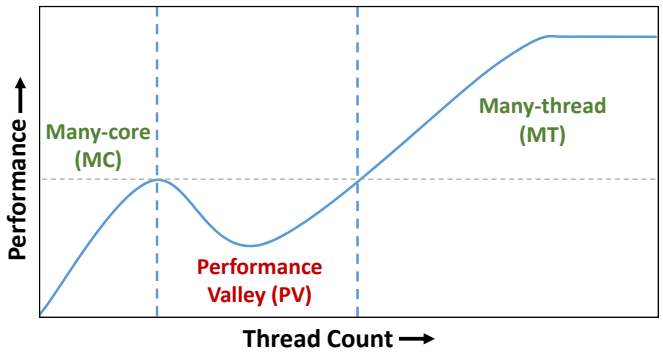


Fig. 1: Performance impact on varying degrees of thread-level parallelism [1]. The MC region represents systems with small number of threads having large per-thread cache capacity to hide latency. Systems in the MT region however leverage massive multithreading as a way to achieve latency tolerance.

To enhance GPU caching efficiency, recent studies [2], [3], [4] have proposed to limit the amount of thread-level parallelism (TLP) available so that cache contention is minimized by reduced sharing. These throttling techniques tune the total number of concurrently executing threads to balance overall throughput and cache miss ratio, such that overall performance is improved by having the active threads climb out of the "performance valley" (into region MC in Figure 1). As we demonstrate in this paper, however, thread throttling leaves other resources under-utilized, such as issue bandwidth, interconnection network bandwidth, and memory bandwidth.

In this paper, we present a fundamentally new approach to the on-chip caching mechanism that is tailored to a massively multithreaded, throughput computing environment. We present *priority-based cache allocation* (PCAL) that improves upon previous thread throttling techniques and enables higher throughput and better shared resource utilization. Prior thread-throttling studies [2], [3], [4] require a sophisticated microarchitecture to dynamically estimate the optimal level of throttling for improved performance. The key goal of our proposal is not only to address the cache thrashing problem but also to develop a lightweight, practical solution that can potentially be employed in future GPUs. Unlike pure thread throttling approaches which force all active threads to feed the on-chip cache, PCAL categorizes active threads into *regular* and *non-polluting* threads. The non-polluting threads are unable to evict data from on-chip storage that has been touched by regular threads. Such an approach reduces the cache thrashing

§ *First authors Li and Rhu have made equal contributions to this work and are listed alphabetically. Li and Rhu were students at the University of Texas at Austin when this work was done and are now with Qualcomm and NVIDIA, respectively. This research was, in part, funded by the U.S. Government and gifts from Microsoft Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.*

problem while, at the same time, effectively allows the non-polluting threads to use shared resources that would otherwise go unused by a pure throttling mechanism. Compared to pure thread-throttling, PCAL allows a larger number of threads to be concurrently executing, while still being able to avoid cache thrashing. In other words, our proposal avoids the "performance valley" by moving the regular threads into the caching-efficient MC region while at the same time leveraging the extra non-polluting threads for extra throughput (similar to the effect of staying in the MT region), a key insight we detail in the rest of this paper. To summarize our most important contributions:

- This paper provides a cost-effective solution to the cache thrashing problem in a massively multithreaded processor environment. While previous studies proposed to stay away from the "performance valley" by throttling the number of threads that time-share the cache, we argue that it is still possible to achiveve similar goals by strictly categorizing threads as being either regular or non-polluting threads. Compared to thread-throttling, our approach achieves both high caching efficiency and high thread-level parallelism.
- We quantitatively demonstrate the above insight by providing a detailed analysis of the primary inefficiencies caused by previous throttling techniques. We observe that simply reducing the number of active threads for better caching efficiency often leads to significant under-utilization of various shared resources across the system architecture.
- Based on this analysis, we propose our novel, lightweight PCAL mechanism that minimizes cache thrashing while maximizing throughput and resource utilization. This paper highlights the importance of balancing TLP, caching efficiency, and system resource utilization rather than simply focusing only on the level of throttling. In summary, our best performing PCAL configuration improves performance by an average 17% with maximum 51% over previous thread-throttling techniques.

II. Background and Related Work

As we conduct our study in the context of GPUs, this section provides background on GPU architectures and its programming model. However, our approach is suitable to any throughput architecture that employs massive multithreading to tolerate system latencies.

A. CUDA Programming Model

Current GPUs, such as those from NVIDIA [5], consist of multiple shader cores (streaming multiprocessors or SMs in NVIDIA terminology), where each SM contains a number of parallel lanes that operate in SIMD fashion: a single instruction is issued to each of the parallel SIMD lanes. In CUDA, a single program *kernel* is executed by all the threads that are spawned for execution. The programmer groups the threads into a *cooperative thread array* (CTA), where each CTA consists of multiple *warps*. Current SMs schedule the instruction to be executed in a warp of 32 *threads*, yet the SIMD grouping of warps is not generally exposed to the CUDA programmer. The number of threads that form a CTA is an application parameter, and a CTA typically consists of enough threads to form multiple warps. As such, programmers are generally encouraged to expose as much parallelism as possible to maximize thread-level parallelism and latency tolerance. Other studies [2], [6], however, point out that maximizing parallelism does not necessarily lead to highest performance. We detail such related works in Section II-D.

TABLE I: Per-thread cache capacity of modern processors.

NVIDIA Kepler GK110 [5]	**Intel Core i7-4960x**	**IBM Power7 [12]**	**Oracle UltraSPARC T3 [13]**
48 KB L1	32 KB L1	32 KB L1	8 KB L1
2,048 threads/SM	2 threads/core	4 threads/core	8 threads/core
24 B/thread	16,384 B/thread	8,192 B/thread	1,024 B/thread

B. Contemporary GPU Architecture

Along with multiple SMs, current GPUs contain a shared on-chip L2 cache and multiple high-bandwidth memory controllers. NVIDIA's high-end GK110, for instance, has 15 SMs, 1.5MB of on-chip L2 cache, and 6 GDDR5 memory controllers [7]. Each SM includes a 128 KB register file, many parallel arithmetic pipelines, a 64KB local SRAM array that can be split between an L1 cache and a software controlled scratchpad memory, and the capacity to execute up to 2,048 threads. In this model, the largest L1 cache that can be carved out of the local SRAM is 48KB.

C. GPU Cache Hierarchy and Its Challenges

GPUs and CPUs are fundamentally different in how they expose parallelism and tolerate memory latency. GPUs rely on massive multithreading to hide latency and have historically used on-chip caches as bandwidth filters to capture stream-based spatial locality and reduce pressure on off-chip bandwidth. CPUs have historically employed much more limited threading and rely on large on-chip cache hierarchies to capture the working sets of applications. This difference is highlighted in Table I, which summarizes the thread and cache capacity of contemporary parallel processing chips. The primary cache capacity per thread for CPUs is 2–3 orders of magnitude larger than for a GPU. As a result, keeping the working set of all threads resident in the primary cache is infeasible on a fully-occupied SM; the GPU is forced to rely on either more plentiful registers, programmer managed scratchpad, or off-chip memory for working set storage. Due to the limited effectiveness of on-chip caching, some GPU devices disable or limit caching capability by default [8], and some GPU cache studies have pointed out that limiting access to the cache can improve performance for a set of applications [9], [10]. However, other studies have demonstrated that the cache hierarchy in many-core processors is still essential for good performance and energy efficiency for some applications [11], [2], [3], [4], which we further discuss below.

D. Related Work

Recent studies have shown that massive multithreading can cause significant cache conflicts. Guz et al. [1] [14]

describe a "performance valley" that exists between systems in which the working set of the active threads fits primarily within the cache and systems with massive multithreading where fine-grained context-switching can hide the memory latency. Several variants of thread throttling mechanisms have been proposed to climb out of this "valley", hence moving the threads to the larger per-thread capacity domain. These proposals seek to reduce the number of threads executing and competing for the on-chip cache. Rogers et al. [2] proposed *cache-conscious wavefront scheduling* (CCWS) which reduces the number of warps that are active and able to execute, such that the number of threads competing for the cache is reduced and cache hit rate is improved. The authors of this work discuss two thread-throttling strategies. First, a *static warp limiting* (CCWS-SWL) approach is proposed which relies on the programmer to manually fix the number of active warps at kernel launch, having a very low implementation overhead but with a higher burden on the programmer to determine optimal warp count. Secondly, a *dynamic CCWS mechanism* that trades off low programmer burden with additional hardware overhead (e.g., 5-KB victim tag array per SM to estimate lost-locality) is presented which approximates optimal warp count based on runtime feedback. In effect, CCWS balances TLP and caching efficiency, improving overall performance. Because of this reactive nature, dynamic CCWS is reported as not being as performant as the static CCWS-SWL approach which identifies the optimal level of TLP before a kernel begins and maintains that optimal level throughout kernel execution. In response to this performance gap, Rogers et al. [4] subsequently proposed an alternative mechanism that proactively throttles the number of executable threads, demonstrating that such a dynamic mechanism can match and at times slightly exceed the benefits of the statically optimal level of TLP. Kayiran et al. [3] proposed a similar dynamic throttling mechanism, with a key difference being that the granularity of TLP adjustment is in CTAs rather than in warps as in CCWS. As with our PCAL mechanism, these thread-throttling approaches similarly focus on assigning different execution capability in a per-warp granularity so we quantitatively detail its benefits and its limitations in Section V, followed by a comparison against PCAL in Section VII.

In addition to the thread-throttling based approaches, below we discuss other studies that are orthogonal to PCAL and can be adopted on top of our proposal for further performance improvements. Several researchers have proposed a variety of schedulers that preferentially schedule out of a small pool of warps [15], [16]. These two-level schedulers have been developed for a number of reasons, but all of them generally have the effect of reducing contention in the caches and memory subsystem by limiting the number of co-scheduled warps. Apart from these warp scheuling based proposals, Jia et al. [10] describe a compile-time algorithm to determine whether to selectively enable allocation in the L1 data cache for each load instruction in the program. The algorithm focuses on the expected degree of spatial locality among the accesses within a warp. Accesses that are anticipated to require many cache lines to satisfy are selectively marked to bypass the L1 data cache, thereby reducing cache capacity pressure. Jia et al. [17] subsequently proposed a hardware-based scheme that provides *request-reordering* and *bypass-on-stall* mechanisms to alleviate the cache contention issues within GPUs. Based on the observation that massive multithreading frequently interrupts intra-warp reuse patterns, the authors proposed to adopt per-warp queues to hold and group memory requests within the same warp such that intra-warp locality is preserved. The bypass-on-stall technique allows a memory request to bypass the cache when all cache blocks within the corresponding cache set and MSHR entries have been already reserved for previous pending requests. This policy hence prevents earlier pending requests from blocking the current request set-conflict issues, achieving higher memory-level parallelism and performance. In general, proposals from [10], [17] can be applied on top of PCAL for higher caching efficiency as it provides a fine-grained, per cache-line control over cache allocation decisions – as opposed to thread-throttling and PCAL which are more about applying different execution privileges in a per-warp granularity.

III. Key Idea of Priority-based Cache Allocation

As described above, previous techniques have attempted to stay out of the "performance valley" by adjusting either the degree of multithreading or the ability for different threads to contend for the cache. Our scheme seeks to holistically incorporate both concepts to find the right balance between thread-level parallelism, cache hit rate, and off-chip memory bandwidth utilization in order to maximize performance.

If we consider a simple throttling scheme in which we limit the number of co-executing threads, we may increase cache hit rate and overall performance over the baseline with maximum TLP enabled. At the same time, we may be introducing vacant instruction issue slots and reducing off-chip memory bandwidth utilization due to the reduced TLP. We propose to allow additional threads to execute that are unable to pollute the cache, but which can take advantage of the available execution and main memory bandwidth, increasing performance over a pure thread throttling approach.

Our priority-based cache allocation (PCAL) scheme seeks to identify a number of threads which can allocate data in the cache, giving them the "right to cache" via a *priority token*. These regular, token-holding threads roughly correspond to the pool of threads we would identify in a basic thread throttling scheme. We also identify an additional number of *non-polluting* threads that can exploit the remaining under-utilized execution and memory bandwidth in the machine. By independently selecting a number of threads that are able to compete for cache resources and a number of additional threads that are unable to pollute these cache resources, we aim to maximize performance by simultaneously keeping execution resources busy, maximizing cache hit rate, and exploiting the available main memory bandwidth. In effect, our approach seeks to strictly move the threads to either side of the "performance valley" (Figure 1) and prevent them from being captured in the middle ground, allowing high TLP while mitigating the memory system contention issues that affect these highly-threaded systems. Because PCAL grants tokens in a per-warp granularity, it is simple to implement (Section VI-E) while also enabling further extensions that can optimize the cache allocation decisions in a finer granularity [17]. Section VI details our key insights and the architecture implementation of PCAL.

TABLE II: Baseline GPGPU-Sim configuration.

Number of SMs	15
Threads per SM	1536
Threads per warp	32
SIMD lane width	32
Registers per SM	32768
Shared memory per SM	48KB
Schedulers per SM	2
Warps per schedulers	24
Warp scheduling policy	Greedy-then-oldest [2]
L1 cache (size/assoc/block size)	16KB/4-way/128B
L2 cache (size/assoc/block size)	768KB/16-way/128B
Number of memory channels	6
Memory bandwidth	179.2 GB/s
Memory controller	Out-of-order (FR-FCFS)

TABLE III: Cache-sensitive CUDA benchmarks.

Abbreviation	Description	Ref.
KMN	K-means	[2]
GESV	Scalar-vector-matrix multiply	[19]
ATAX	Matrix-transpose-vector multiply	[19]
BICG	BiCGStab linear solver sub-kernel	[19]
MVT	Matrix-vector-product transpose	[19]
CoMD	Molecular Dynamics	[24]
BFS	Breadth first search	[23]
SSSP	Single-source shortest paths	[23]
SS	Similarity Score	[21]
II	Inverted Index	[21]
SM	String Match	[21]
SCLS	Streamcluster	[22]

IV. Methodology

Benchmarks. We evaluate a large collection of GPU benchmark suites from NVIDIA SDK [18], PolyBench [19], Parboil [20], Mars [21], Rodinia [22], LonestarGPU [23] and CoMD [24]. Due to long simulation times, we execute some of these applications only until the point where performance exhibits small variations among different iterations of the kernel. We identify *cache-sensitive* benchmarks in the following manner. We first simulate each benchmark with the baseline configuration in Table II and again with L1 and L2 cache capacities increased by 16×. We then classify an application as cache-sensitive if the large cache configuration achieves a speedup of 2× or more. Table III summarizes the 12 cache-sensitive benchmarks that we focus in the rest of this study.

Simulation Methodology. We model the proposed architecture using GPGPU-Sim (version 3.2.1) [25], [9], which is a cycle-level performance simulator of a general purpose GPU architecture supporting CUDA 4.2 [26] and its PTX ISA [27]. The GPU simulator is configured to be similar to NVIDIA GTX480 [28] using the configuration file provided with GPGPU-Sim [29] (Table II). Additionally, we enhance the baseline GPGPU-Sim model with a XOR-based cache set-index hashing [30] to improve the robustness of the memory system and provide a strong baseline architecture[1]. The modified set-index hashing algorithm is applied at both L1 and L2 caches to better distribute memory accesses among cache banks, mitigating the effect of bank conflicts, and reducing *bank camping* situations where regular access patterns produce excessive contention for a small subset of cache banks.

V. Motivation and Application Characterization

To compare our PCAL scheme with the most relevant state-of-the-art caching optimization strategies, this section analyzes the effect of applying thread throttling [2], [3], [4] and *Bypass-on-Stall* [17] to the baseline architecture of Table II.

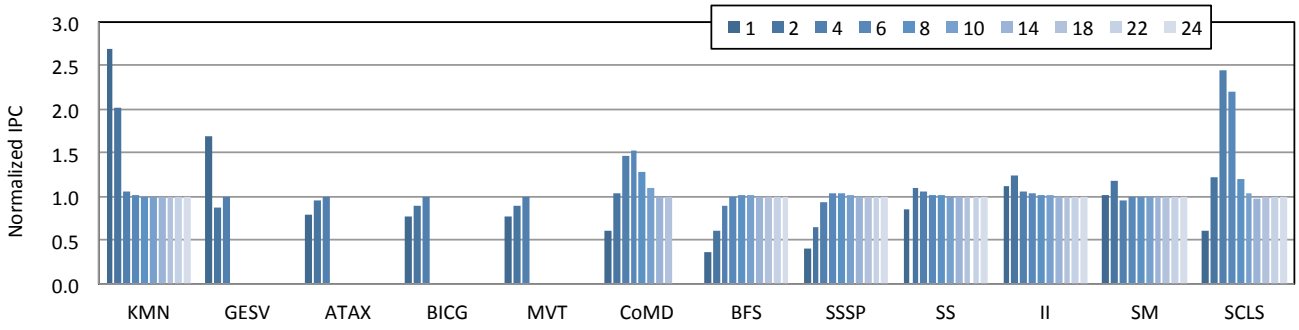
A. Thread Throttling

Although GPU programming models [26], [31] encourage the programmer to expose a large amount of parallelism to maximize latency tolerance, mapping the maximum amount of parallel work that the hardware can support often leads to cache thrashing and significant contention in shared resources. Thread throttling is an effective mechanism to balance TLP and caching efficiency, thereby improving performance. However, limiting the number of threads can potentially leave other shared resources underutilized. To demonstrate how TLP throttling affects various aspects of the system architecture, this section details the effect of TLP modulation on overall performance, L1 miss rate, and off-chip bandwidth utilization.

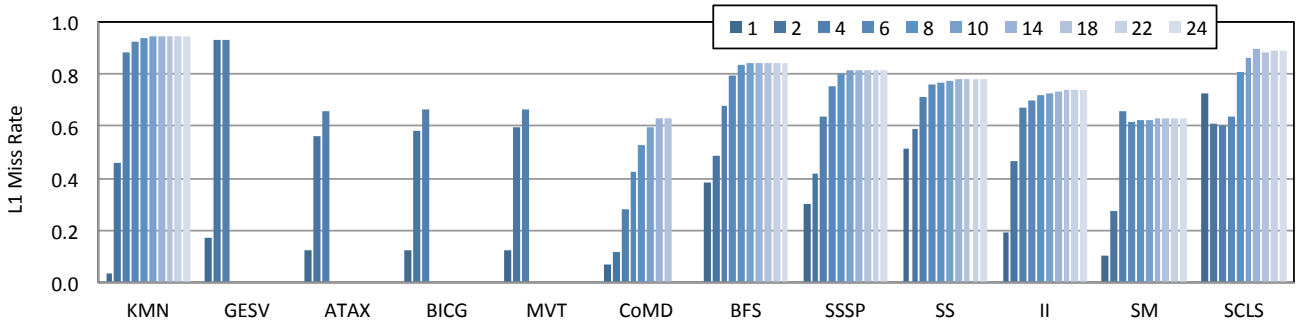
Throttling Granularity. As discussed in Section II-D, Rogers et al. [2] and Kayiran et al. [3] both proposed thread-throttling, the key difference between them being the granularity of throttled threads. Our simulation results for CTA-level throttling was shown to exhibit worse performance than the finer-grained warp-level throttling (e.g., for KMEANS, GESV, etc) because it was unable to reach the optimal number of active warps unless the optimal warp count is a multiple of the CTA size. Since the key intuitions behind the warp-level throttling of [2] and the CTA-level throttling of [3] are similar (i.e., throttle the number of threads when the cache thrashes), we choose the warp-level throttling as a baseline comparison point due to its finer-granularity and flexibility.

Static vs Dynamic Throttling. Rogers et al. [2] discussed *static warp limiting* (CCWS-SWL) to motivate the dynamic CCWS mechanism, reporting that CCWS-SWL always matched or slightly outperformed the dynamic CCWS. While choosing the optimal number of static warp count requires manual performance tuning from the programmer-end, it is a common practice frequently used in the real world application tuning process. Compared to dynamic CCWS, CCWS-SWL requires minimal implementation overhead while outperforming dynamic CCWS. For this reason, we adopt CCWS-SWL as the baseline throttling mechanism to compare against.

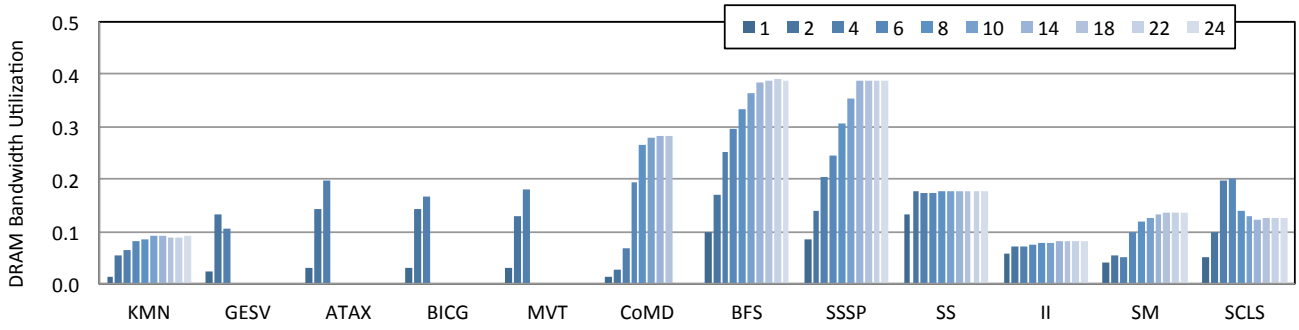
[1]The original cache set-index function of GPGPU-Sim is based on a simple bit-extraction of a subset of the data address. Our evaluation shows that some applications exhibit significant pathological performance (up to 242% slowdown for SRAD in Rodinia [22]) without the more realistic XOR-based set-index hashing baseline [30]. We therefore adopt a simple XOR of bits from the upper address region with the original set index bits.



(a) Overall performance.



(b) L1 cache miss rate.



(c) Off-chip DRAM bandwidth utilization.

Fig. 2: Impact of TLP throttling on performance, L1 cache miss rate, and off-chip bandwidth utilization. Performance is normalized to `warp-max` and the numbers 1, 2, ..., 24 represent the number of warps the thread scheduler is able to issue instructions from. Note that GESV, ATAX, BICG, and MVT contains only three bars as `warp-max` equals 4 warps.

B. Impact of Thread Throttling

TLP throttling is achieved by strictly limiting the number of warps available to the thread scheduler. The numbers 1, 2, 4, ..., 24 in Figure 2 designate the number of warps from which the thread scheduler is able to issue instructions. The number 1, for instance, represents a configuration with minimum TLP available whereas 4 for GESV, 18 for CoMD, and 24 for KMN represents the maximum TLP exposed to the scheduler. The maximum number of CTAs (and hence warps) that can concurrently execute is determined by the number of registers allocated for each thread and the capacity of scratchpad storage allocated for each CTA. Each application therefore exhibits different levels of maximum possible TLP. KMN, for example, can allocate up to 24 warps within a warp scheduler, while GESV can only manage up to 4 warps due to resource constraints.

Performance. Figure 2a shows the performance variation as the number of active warps changes. For many applications, the maximum TLP does not result in the best performance. For instance, CoMD performs best with only 6 warps enabled as it optimally balances overall throughput with caching efficiency, leading to the best performance. On the other hand, KMN exhibits its best performance with only a single warp activated. As KMN inherently contains high data locality, the performance benefits of better caching efficiency (thanks to one active warp) outweighs its potential loss in throughput and latency tolerance. Overall, 9 out of the 12 cache-sensitive benchmarks are shown to benefit from throttling. Table IV summarizes each application's *maximum* number of warps executable (`warp-max`) and the *optimal* number of warps that maximizes overall performance (`warp-CCWS`).

Cache Miss Rate. Figure 2b shows that the benchmarks

TABLE IV: Each benchmark's warp-max, number of optimal warp count for CCWS-SWL (warp-CCWS), and speedup of CCWS-SWL compared to warp-max.

Benchmark	warp-max	warp-CCWS	Speedup
KMN	24	1	2.68
GESV	4	1	1.69
ATAX	4	4	1
BICG	4	4	1
MVT	4	4	1
CoMD	18	6	1.53
BFS	24	8	1.02
SSSP	24	8	1.05
SS	24	2	1.11
II	24	2	1.24
SM	24	2	1.18
SCLS	24	4	2.44

generally exhibit significant reduction in L1 misses with lower TLP. The miss rate of KMN, for instance, dramatically decreases from 94% to 4% with optimal throttling, thanks to less cache contention. However, best performance achieved with CCWS-SWL does not necessarily lead to the lowest cache miss rate. CoMD performs best with 6 active warps, but the best caching efficiency is achieved with only a single warp activated. Such behavior highlights the importance of balancing caching efficiency with TLP, which is effectively achieved with statically throttling TLP with CCWS-SWL.

Off-Chip Bandwidth Utilization. Figure 2c shows the effect warp throttling has on off-chip bandwidth utilization. The off-chip bandwidth utilization is derived by measuring the fraction of DRAM clock cycles when the data bus is busy transferring requested data. For most applications, the bandwidth utilization increases with higher TLP. First, higher TLP leads to more outstanding memory requests and requires more memory bandwidth. Second, higher TLP results in the higher cache miss rates, which eventually leads to more data being fetched from the off-chip DRAM. Overall, among the 9 applications that exhibit higher performance with thread throttling, memory bandwidth utilization is significantly reduced with CCWS-SWL (e.g., CoMD with a 33% reduction and II with a 62% reduction compared to warp-max).

C. Bypass-on-Stall

The bypass-on-stall [17] (BoS) policy has some common aspects to PCAL in that it directs certain cache accesses to *bypass* the cache rather than waiting for previous requests to finish. When previous requests have already reserved all the cache blocks within the corresponding cache set or MSHR entries (which occurs when cache access intensity spikes and congests the resource allocation process), BoS allows incoming requests to make forward progress rather than waiting for these previous requests to complete. Because BoS is an orthogonal approach to thread-throttling, we study the performance benefits of applying BoS at L1 and CCWS-SWL on top of the baseline warp-max. As illustrated in Figure 3, BoS provides an average 6% speedup compared to warp-max with maximum 68% for SCLS, thanks to its ability to avoid cache-set contention. When BoS is applied on top of CCWS-SWL, however, the performance improvements become limited with

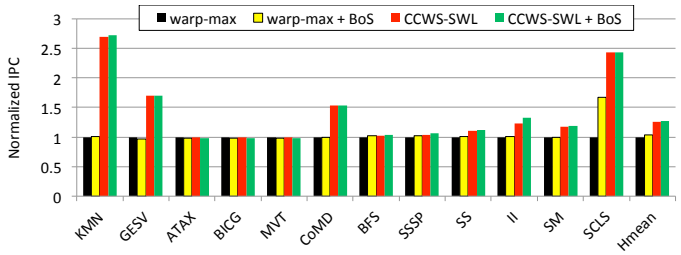


Fig. 3: Performance benefits of CCWS-SWL [4] and BoS [17]. Note that CCWS-SWL represents the highest bar in Figure 2a.

an average 1% (maximum 6% for II) because of the following reasons. First, the more robust cache set-indexing hashing function [30] we applied on top of baseline (Section IV) improves caching efficiency for multiple of these benchmarks, reducing the cache congestion problem in the first place. Second, CCWS-SWL also alleviates cache resource congestion due to the reduction in TLP, making BoS less effective compared to when it is added on top of warp-max. Note that Jia et al. [17] discuss a separate request-reordering technique as to optimize intra-warp reuse patterns, both of which are orthogonal to our proposed PCAL. Studying the added benefits of BoS and request-reordering on top of PCAL is beyond the scope of this paper and we leave it for future work.

D. Limitations

Thread throttling relies on tuning a single parameter, the total number of concurrently executing warps, to balance overall TLP, caching efficiency, and shared resource utilization in the memory system. As discussed in Section V-A, however, such a one-dimensional optimization strategy can lead to under-utilization of those shared resources (e.g., memory bandwidth utilization), leaving performance opportunities on the table. When combined with a better cache set-index hashing and optimal static warp-level throttling (CCWS-SWL), the BoS policy becomes less effective in providing robust performance benefits as CCWS-SWL frequently resolves the cache congestion problem in the first place. Our PCAL mechanism seeks to holistically address the GPU caching problem by minimizing cache pollution while maximizing shared resource utilization.

VI. PCAL Architecture

This section describes the mechanisms for implementing our token-based priority cache allocation (PCAL) scheme. To address caching inefficiencies of massively multithreaded GPU architectures in concert with resource under-utilization due to throttling, we develop PCAL which separates the concerns of cache thrashing and resource utilization. We first provide a high-level overview of PCAL and its architecture design. We then discuss our software-directed, *static* PCAL approach for assigning PCAL parameters on a per-kernel basis, a practical bridge to achieving improved caching utility with minimal implementation overhead. While we use static PCAL to motivate and deliver the key intuition behind our work, we also present our *dynamic* PCAL as a proof-of-concept to demonstrate the feasibility of a dynamic algorithm, alleviating the burden placed to the programmer.

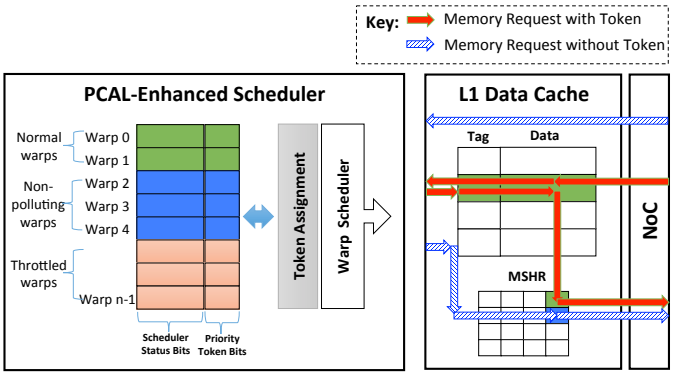


Fig. 4: PCAL architecture.

A. High-Level Overview of PCAL

PCAL assigns priority tokens to warps which grant authorization to perform various cache actions, including allocation (fill) and replacement (eviction). A subset of available warps are assigned tokens and receive prioritized access to the L1 cache. A separate subset of warps may remain executable for enhancing overall TLP, but have limited priority for allocation in the cache. These warps are not able to pollute the cache space. Finally, the remaining warps may be throttled, or temporarily prevented from running, to reduce resource contention. Given that GPU schedulers manage groups of threads in warps, the granularity of token-assignment is also in warps.

Tokens represent priority within various shared resources in the system. While such tokens can be leveraged to orchestrate different access priority to various other resources, this paper focuses on the role of tokens at the L1 cache, the most scarce resource in current GPUs (Table I). Hence tokens represent the ability of warps to execute as normal, with full cache block allocation and replacement privileges in the L1 cache. Threads without tokens remain executable, but are not allowed to induce eviction: in essence, these warps are guaranteed to be non-polluting in the L1 cache.

Figure 4 presents the overall PCAL architecture with the extensions required for token management and token-based cache allocation. The PCAL-enhanced warp scheduler keeps scheduler status bits for each active warp, along with token bits that represent the possession of a token. Based on the token assignment policy, the token assignment unit updates these token bits in the scheduler. When a memory request is generated by the scheduled warp, the warp's token bit is appended to the request before it is sent to the cache-memory hierarchy. The cache control logic, accordingly, uses this token bit to make cache allocation and eviction decisions. We further detail PCAL's various policies and its operation below.

Token Assignment. The total number of tokens may be statically designated by software or dynamically derived at runtime. Our implementation assigns the `T` available tokens to the `T` oldest warps.

Token Release. Once assigned a token, a warp retains the token either until completion or until a barrier is reached. Tokens are released and reassigned at warp termination and assigned to the next oldest warp that does not currently hold a token. Similarly, tokens are also released at synchronization points while a thread waits at a barrier.

Cache Allocation Policy. Token holders access the cache as normal, allocating cache blocks as desired. Token-less warps (non-polluting warps) are not allowed to fill any cache block.

Cache Eviction Policy. Similar to the cache allocation policy, possessing a priority token indicates a warp has permission to initiate replacement (eviction). Because non-token holders are not allowed to induce cache block fills, they cannot induce evictions. However, we do allow non-token holders to update cache LRU bits on access to recognize inter-warp sharing.

Data Fetching Granularity. Current GPU SMs contain a memory coalescing unit that aggregates the memory requests of the active threads within each warp; whose requesting data size can range from 32 to 128 bytes. Rhu et al. [32] highlighted the need for a sectored cache hierarchy in throughput processors so that data fetches do not always fetch a full cache-line when only a portion of it is needed. Our PCAL mechanism incorporates some of the key insights of this prior work by having a non-polluting warp's L1 missed data be directly forwarded from the L2 to the SM in an on-demand fashion, rather than fetching the entire L1 cache-line. Such on-demand forwarding enables efficient bandwidth utilization in the interconnection network, giving more headroom for modulating TLP using PCAL.

B. PCAL Performance Optimization Strategies

To optimally balance TLP, caching efficiency, and off-chip bandwidth utilization, we consider two basic strategies to maximize overall performance when employing PCAL.

Increasing TLP while Maintaining Cache Hit Rate (ITLP). After thread throttling is applied, the remaining active warps might not be sufficient to fully saturate the shared resource utilization, giving headroom to add additional non-polluting warps. In this case, PCAL first hands out tokens to all remaining warps and adds the minimum number of extra non-polluting warps to saturate such shared resources (e.g., interconnection and memory bandwidth). The caching efficiency of the normal warps are generally not disturbed with this strategy as the token-less warps do not have authorization to pollute the cache. These extra non-polluting warps enable PCAL to utilize chip resources more effectively and get extra throughput beyond that provided by pure thread throttling.

Maintaining TLP while Increasing Cache Hit Rate (MTLP). In some cases, the shared resources are already saturated so spare resources are not sufficient to add extra non-polluting warps. Additionally, as some applications simply achieve the highest throughput with maximum TLP, no extra non-polluting warps can be added. In these cases, PCAL maintains the current total number of active warps and reduces the number of tokens that can be handed out. Only the subset of warps that are assigned tokens can keep their working sets inside the cache. Since the remaining non-token holding warps cannot pollute the cache, overall caching efficiency is improved while not reducing TLP.

C. Static Priority-based Cache Allocation

Expert GPU programmers often put significant effort to optimally tune various kernel parameters so that overall per-

TABLE V: Microarchitectural configuration for dynamic PCAL.

Sampling period	50K cycles
IPC variation threshold	20%
L1 miss-rate variation threshold	20%
ITLP-threshold for L2-queuing-delay	15 cycles
ITLP-threshold for NoC-queieing-delay	20 cycles
ITLP-threshold for MSHR-rsv-fauilure-rate	70%

formance is maximized. For instance, it is a common practice to manually tune the CTA size or the amount of scratchpad memory allocated per CTA to modulate the level of TLP for maximum performance. We present software-directed, static PCAL as an additional tool that performance-conscious developers can leverage. Static PCAL requires minimal implementation overhead and can be integrated easily into both current hardware and the software stack. Given that the range of applications that can practically benefit from thread-throttling is fairly limited, a static approach reduces the barrier for adoption by GPU vendors. It also provides the underlying mechanisms as a first step required to evolve (via hardware and/or software) towards more automated or programmer-friendly approaches for parameter or policy selection.

Software-directed PCAL allows the programmer to specify the following parameters to the warp scheduler: (1) the maximum runnable warps (`W`), and (2) the number of warps that are assigned priority *tokens* (`T`). When `T` is less than `W`, the remaining (`W-T`) warps remain executable but do not receive tokens and are hence non-pollutable. The input parameters (`W`, `T`) are supplied at launch time with other kernel parameters and may be selected by the developer, an automated compiler, or by an autotuner. Such statically designated parameters are fed to the warp scheduler, where warps are divided into three categories: (1) *throttled* – not eligible to execute, (2) *executable with tokens* – high priority with right to allocate cache-lines, and (3) *executable without tokens* – no caching privileges. Tokens are represented with an additional bit and the token assignment unit within the warp scheduler is responsible for allocating and managing tokens. The token bit is appended to the memory request packet and to the per-line storage of priority token meta-data for the L1 cache. The cache control logic uses the token bit of the issuing warp to determine if the request may allocate space in the cache.

D. Dynamic Priority-based Cache Allocation

While static PCAL is an effective tool for performance tuning GPU applications, finding the optimal parameter values for maximum performance requires additional developer effort. This section describes the two versions of our dynamic mechanism for PCAL to determine PCAL parameters automatically at runtime. We first discuss our lightweight, MTLP-based dynamic PCAL mechanism which achieves competitive speedups compared to CCWS while requiring minimal hardware overheads, as opposed to CCWS which requires 5-KB of SRAM tag array per SM [2]. We then present a more aggressive version of dynamic PCAL that leverages both MTLP and ITLP strategies, showing comparable performance improvements as the best performing static PCAL configuration.

Algorithm 1 History table (HT) update and reload operation.

```
reload_W_T_count() {
if (HT[kernel_idx].valid==false) or (HT[kernel_idx].confidence<0) then
    (W, T) = default;
else
    (W, T) = HT[kernel_idx].best_W_T;
end if
}

update_history_table() {
if (HT[kernel_idx].valid==false) then
    HT[kernel_idx].best_W_T = current_best_W_T;
    HT[kernel_idx].past_IPC = current_IPC;
    HT[kernel_idx].confidence = 2;
else if (HT[kernel_idx].past_IPC<=current_IPC) then
    HT[kernel_idx].confidence ++
else
    HT[kernel_idx].confidence --
end if
}
```

Dynamic PCAL with Max TLP. Our lightweight dynamic PCAL mechanism ($\texttt{dynPCAL}_{MTLP}$) adopts only the MTLP strategy to reduce the two-dimensional (`W`, `T`) search space down to a single-dimension. $\texttt{dynPCAL}_{MTLP}$ first launches the kernel with the total number of warps (`W`) at all SMs fixed at maximum TLP (`warp-max`), but the number of tokens (`T`) available to the individual schedulers are varied across different SMs. After executing for a given sampling period, the performance impact of different numbers of tokens is evaluated by a *parallel voting mechanism*. At the end of this sampling period, the token count that led to the highest performance is chosen for adoption by all the SMs and each continue execution under the same number of tokens. While executing, if any of the SMs experience more than a predefined threshold IPC drop, $\texttt{dynPCAL}_{MTLP}$ initiates another round of parallel voting to reach a common number of tokens with higher performance. The first three entries in Table V summarizes the key parameters used for evaluating $\texttt{dynPCAL}_{MTLP}$ (we discuss the usage of the last three parameters later in this section).

Because $\texttt{dynPCAL}_{MTLP}$ primarily relies on existing performance counters within each SMs, the major overhead is in implementing a simple finite-state-machine that manages the parallel voting process. As we detail in Section VII-B, $\texttt{dynPCAL}_{MTLP}$ provides significant performance speedup compared to `warp-max` despite its low overhead.

Dynamic PCAL with Throttling. While $\texttt{dynPCAL}_{MTLP}$ presents a low-cost solution that provides a majority of the benefits of CCWS, it still falls behind the best performing static PCAL configuration, primarily because it does not exploit the ITLP strategy. As $\texttt{dynPCAL}_{MTLP}$ fixes the total number of warps at `warp-max`, it is unable to adjust the total amount of TLP. As a more aggressive design point of dynamic PCAL, we discuss the impact of enabling both ITLP and MTLP strategies for maximum performance. To evaluate the efficacy of both strategies, this aggressive version of dynamic PCAL initially throttles down the number of executable warps to less than `warp-max` such that the ITLP strategy can be applied. The number of warps to throttle can be directed either by an autotuner, static compiler analysis [10], or a CCWS-like hardware microarchitecture [2] that dynamically estimates the optimal level of throttling. This paper assumes it is augmented on top of CCWS to demonstrate its feasibility ($\texttt{dynPCAL}_{CCWS}$). Both the MTLP strategy of $\texttt{dynPCAL}_{MTLP}$ and the ITLP

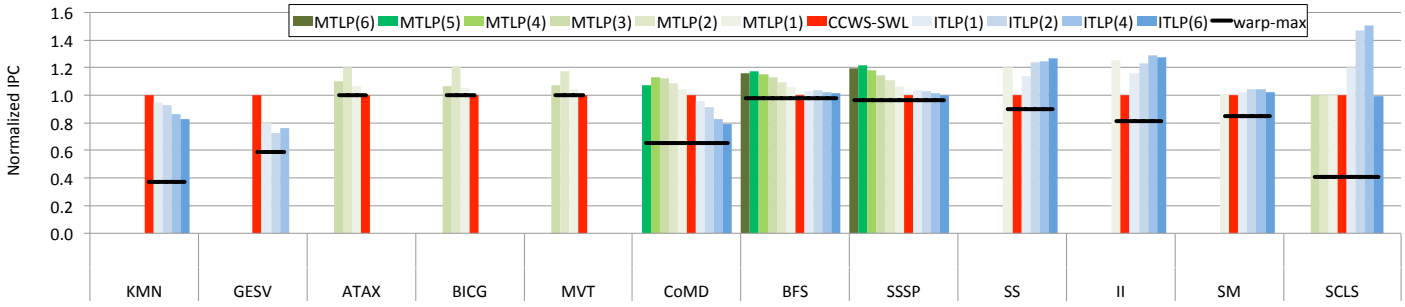
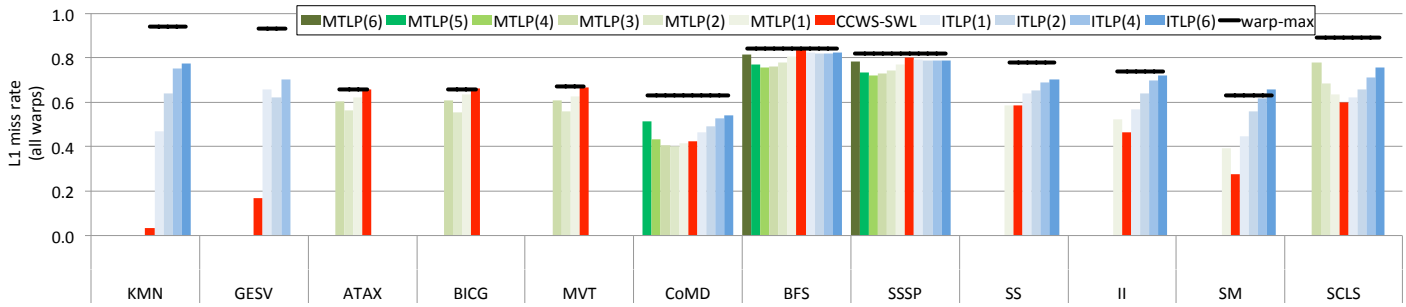
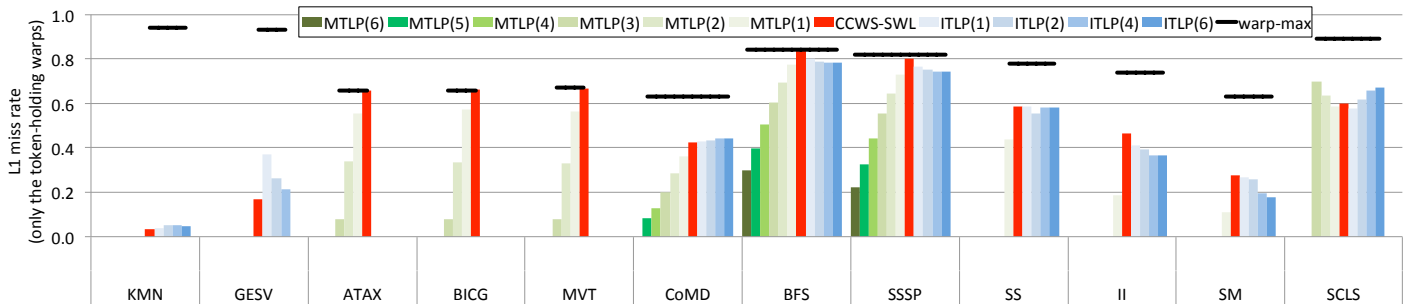


Fig. 5: Performance (normalized to CCWS-SWL).



(a) L1 miss rate for all active warps.



(b) L1 miss rate excluding the non-polluting warps.

Fig. 6: L1 miss rate.

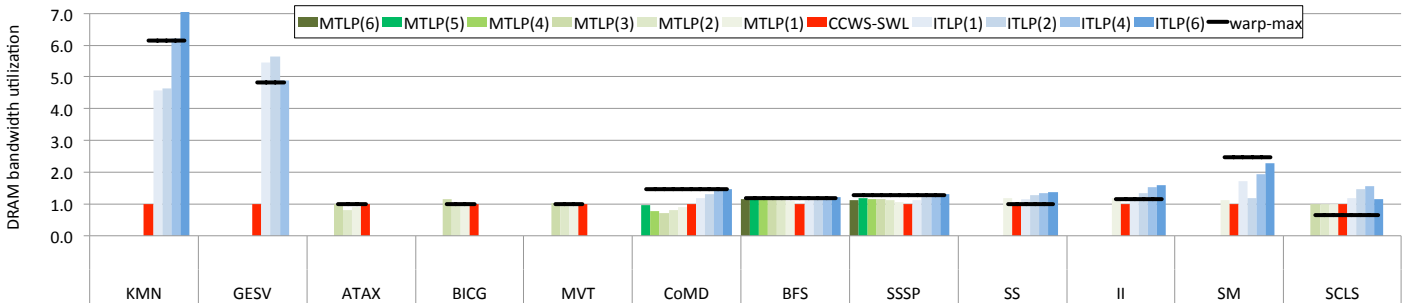


Fig. 7: Off-chip memory bandwidth utilization.

strategy (detailed below) are evaluated to choose the best performing (W, T) combination. For MTLP, we employ the $\texttt{dynPCAL}_{MTLP}$ approach to determine the best token number through parallel voting. Dynamic ITLP on the other hand is initiated by monitoring the level of various on-chip shared resource utilization, including queueing delays at the network-on-chip (NoC) and the L2 tag lookups, MSHR reservation failure at the L2. When the ITLP control unit (implemented as a finite-state-machine) observes that all the shared resource utilization is below a given threshold (the last three parameters in Table V), all SMs add an additional non-polluting warp to examine the effects of added TLP during the next sampling

period. If the SM achieves more than a threshold amount of IPC improvement, one additional non-polluting warp is added again to the scheduler to futher saturate the shared resource utilization. If the SM experiences more than a threshold level of IPC loss, however, a single non-polluting warp is prevented from begin scheduled so that resource congestion is reduced. As in MTLP, ITLP's simple *hill-climbing* approach leverages existing (or a small number of additional) performance counters.

For applications that undergo multiple iterations of the kernel, it is possible that the optimal (W, T) will be same (or similar) across different iterations. To minimize the searching overheads of MTLP/ITLP, we augment dynPCAL$_{CCWS}$ with a simple *history table* that stores prior history of best performing (W, T) information. Since different executions of the kernel can also exhibit varying phase behavior, each table entry is augmented with a confidence field that reflects the reliability of the given entry. Algorithm 1 summarizes how dynPCAL$_{CCWS}$ manages the history table and how the confidence field is used to react to different phasing behavior of a given kernel.

E. Implementation Overhead

We implement the possession of a token as a single bit, so the thread scheduler entry needs to track this additional bit. A small amount of logic is required to allocate and manage tokens. Additionally, memory requests need to be tagged with this additional information. Minimal logic overhead is required for managing the assignment of tokens. Overall, we expect the overhead of static PCAL to be minimal.

dynPCAL$_{MTLP}$ requires a handful of logic for the finite-state-machine controlling the MTLP optimization process. Aside from the state-machine to implement ITLP/MTLP control units, dynPCAL$_{CCWS}$ only requires an additional 8-entry fully-associative history table, each entry containing 9 bits for the kernel index, 10 bits for (W, T), and 10 bits for storing prior IPC (Algorithm 1). The single table is shared across all SMs. Overall, we estimate the hardware overheads of dynamic PCAL to be negligible.

VII. Results and Discussion

This section evaluates PCAL's impact on cache efficiency, off-chip bandwidth utilization, and performance improvements compared to the state-of-the-art (Section V). All average values presented in this paper are based on harmonic means.

A. Static Priority-based Cache Allocation

Figure 5 summarizes the performance benefits of static PCAL compared to CCWS-SWL. In the figure, ITLP(N) refers to having the number of token-holding warps identical to CCWS-SWL while adding additional *N* non-polluting warps to increase TLP and better utilize shared resources. MTLP(M) represents our PCAL strategy where the total number of active warps are maintained identically as in CCWS-SWL but those that are able to allocate cache space are reduced by *M*, improving caching efficiency. KMN and GESV are not eligible for the MTLP strategy because CCWS-SWL corresponds to a single active warp, whereas ATAX, BICG, and MVT cannot adopt ITLP as CCWS-SWL equals warp-max (Table IV).

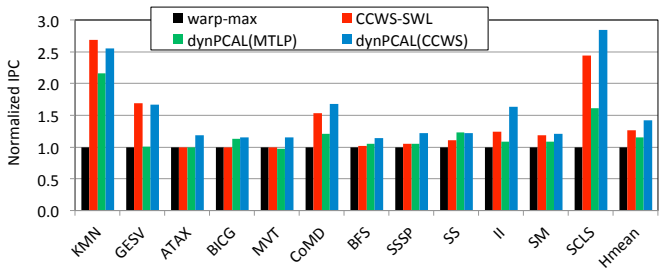


Fig. 8: Speedup of dynPCAL$_{MTLP}$ and dynPCAL$_{CCWS}$ (normalized to warp-max).

Overall, the best performing static PCAL leads to an average 17% performance improvement compared to CCWS-SWL. Six of the twelve cache-sensitive applications exhibit the highest speedup with the MTLP strategy, while four prefer ITLP. The remaining two benchmarks, KMN and GESV, shows no benefits with PCAL. The L1 caching efficiency of KMN and GESV are very sensitive to the the amount of TLP, significantly aggravating overall L1 miss rate when more than a single warp is activated (Figure 6). The six applications amenable to MTLP are those that already congested shared resources with CCWS-SWL, so adding more non-polluting warps for increased throughput has limited effectiveness. Rather, by reducing the number of warps contending for the L1 cache, MTLP improves the caching efficiency of the token-holding warps (Figure 6b) while consuming similar memory bandwidth compared to CCWS-SWL (Figure 7). For the four applications that prefer ITLP (SS, II, SM, and SCLS), the performance benefit primarily comes from higher throughput in the memory system, leading to an average 57% increase in memory bandwidth utilization. Caching efficiency of the token-holding warps is generally preserved, demonstrating the effectiveness of PCAL's ITLP strategy relative to CCWS-SWL.

B. Dynamic Priority-based Cache Allocation

Figure 8 summarizes the performance of our lightweight dynPCAL$_{MTLP}$ compared to warp-max and CCWS-SWL. dynPCAL$_{MTLP}$ achieves an average 18% performance improvement compared to the warp-max baseline, while CCWS-SWL achieves an average 26%. Given its substantially simpler hardware overhead, dynPCAL$_{MTLP}$ presents a cost-effective alternative to CCWS-style dynamic throttling mechanisms. In addition, Figure 9 shows that the dynPCAL$_{CCWS}$ scheme, with the ability to leverage both the ITLP and MTLP strategies, outperforms all warp-max/CCWS-SWL/BoS and provides an average 11% speedup over CCWS-SWL(Figure 9). dynPCAL$_{CCWS}$ also achieves 96% of the throughput of static PCAL on average.

Overall, dynamic PCAL with its simple algorithm and straightforward design, provides a very low-cost technique to replace CCWS (dynPCAL$_{MTLP}$) or enhance CCWS (dynPCAL$_{CCWS}$).

C. Sensitivity Study

PCAL on Cache-Insensitive Workloads. Static PCAL provides a performance tuning knob to the programmer where the

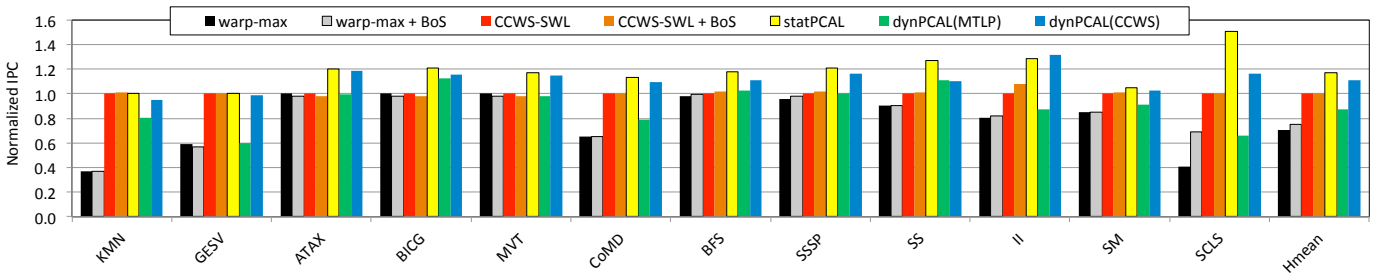


Fig. 9: Speedup of all static/dynamic PCAL (normalized to `CCWS-SWL`).

decision to enable PCAL or not can be provided as a compiler or kernel-launch option. Static PCAL can be disabled (effectively giving tokens to all warps) for insensitive workloads to ensure no performance degradation is caused. Our experiments showed that dynamic PCAL has negligible performance impact for 38 cache-insensitive workloads: less than 1% average performance degradation (with maximum of 3.4%). Dynamic PCAL will disable PCAL for cache-insensitive workloads, but requires a learning period before detecting cache-insensitivity. Because training is short compared to overall execution time, the overhead was minimal.

Microarchitectural Configuration for Dynamic PCAL. We examined the performance sensitivity of dynamic PCAL to the parameters in Table V, but did not observe major performance variations. The reported parameters in Table V are those that provided stable, consistent results across the benchmarks evaluated.

PCAL on L2 caches. This paper primarily focuses on the impact of applying PCAL at the L1. While not detailed in this paper, we also studied the impact of employing PCAL on both L1 and L2. Applying PCAL to both caches results in either negligible performance improvement or a performance penalty, depending on the benchmark. While we observed comparable L1 miss rates for both cases, the L2 miss rate increases from an average of 31% to 44% when adding PCAL to the L2. Employing PCAL for both L1 and L2 leads to a performance degradation because the non-polluting warps allocate in neither the L1 nor the L2. If the L2 is not full of more critical data, preventing non-polluting warps from caching in the L2 leads to a higher L2 miss rate. Even when excluded from the L1, the non-polluting warps see benefit from caching in the L2 rather than being forced to fetch from the off-chip DRAM. Furthermore, as the token-holding warps see improved L1 hit rates from reduced thrashing, they are less likely to access data from the L2, effectively freeing up L2 capacity and bandwidth for non-polluting warps.

D. Discussion and Extensions

Opportunistic Caching. We consider allowing cache blocks to be allocated *opportunistically*, meaning that a non-polluting warp can fill data into the cache if there are empty cache blocks not currently occupied by token holders. In this scheme, PCAL marks cache blocks allocated by token-holding warps with priority bits and allows non-polluting warps to reserve and fill only unmarked cache blocks. To allow more cache blocks to be used opportunistically by non-polluting warps, we can clear the priority bits of each cache block at various intervals. We evaluated several methods for clearing cache block priority bits: (1) clearing when a kernel ends, (2) clearing when all owners terminate, (3) clearing when the last owner terminates, and (4) clearing periodically. Our experiments show that all four methods perform similarly. Consequently, the results reported in this paper assume opportunistic caching is *disabled* due to limited effectiveness.

Token Types. While this paper primarily focuses only on L1 token configurations, other token variations are possible; (1) L2 tokens, which can be assigned to warps mutually exclusive to the L1 token holders; (2) DRAM tokens, which can be assigned to warps not assigned with L1/L2 tokens; and (3) combination of L1/L2/DRAM tokens. We leave the exploration of such token variations for future work.

Token Assignment/Release Policies. Our implementation of PCAL provides tokens to the oldest warps, and tokens are neither shared nor transferred to other warps until the warp terminates or reaches a barrier. This token assignment and release policy may not be desirable for applications that contain different program phases, such as those with a set of warps that only need L1 cache capacity during the part of their lifetime. We leave to future work a study of such sophisticated token assignment and release mechanisms that adapt to program phase behavior.

Power Efficiency. Thanks to its improved caching and performance, PCAL is generally expected to provide better power efficiency. There are however situations in which the most performance-optimal PCAL configuration may not necessarily lead to the optimal power efficiency. The benchmark SM, for instance, achieves its best performance with ITLP(4), but the off-chip bandwidth utilization of ITLP(4) is 64% higher than with ITLP(2). Given the performance advantage of ITLP(4) over ITLP(2) is less than 3%, a more power-efficient approach would execute SM with the ITLP(2) to reduce power-expensive DRAM bandwidth. An investigation of power-optimized dynamic algorithms is left for future work.

VIII. Conclusion

In this paper, we showed that cache-sensitive applications have the opportunity to increase cache locality, but not merely by increasing cache size. Reducing the threads that compete for the cache is the key element to enabling better usage of the cache resources. We propose a priority-based cache allocation (PCAL) mechanism that gives preferential access to the cache and other on-chip resources to a subset of the threads

while having the remaining threads avoid cache pollution by preventing them from cache space allocation. Our novel token-based approach reduces cache contention and effectively employs the chip resources that would otherwise go unused by a pure thread throttling approach.

Our results show that a software-directed PCAL scheme enables a 17% performance improvement over optimal static thread-throttling schemes for cache-sensitive applications. We also demonstrate a low-overhead dynamic algorithm that tunes the PCAL parameters such that it can provide a 23% speedup over the baseline performance (providing a majority of the speedup of a scheme like CCWS without the associated hardware overheads). We show that a dynamic PCAL scheme can supplement thread-throttling approaches like CCWS, providing an additional 11% speedup on top of the CCWS approach. This basic approach of decoupling thread-level-parallelism from cache allocation privileges provides a simple, inexpensive approach to bridge the valley between low-thread-count, high-cache-utility and highly-multithreaded, streaming performance regimes.

REFERENCES

[1] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, January 2009.

[2] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *International Symposium on Microarchitecture (MICRO)*, December 2012, pp. 72–83.

[3] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2013, pp. 157–166.

[4] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *International Symposium on Microarchitecture (MICRO)*, December 2013, pp. 99–110.

[5] NVIDIA Corporation, "Whitepaper: NVIDIA GeForce GTX 680," 2012.

[6] Vasily Volkov, "Better Performance at Lower Occupancy," 2010. [Online]. Available: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

[7] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," NVIDIA White Paper, 2012.

[8] NVIDIA Corporation, "Tuning CUDA Applications for Kepler," July 2013. [Online]. Available: http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf

[9] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[10] W. Jia, K. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-Fetched Caches in GPUs," in *International Symposium on Supercomputing (ICS)*, June 2012, pp. 15–24.

[11] C. Hughes, C. Kim, and Y.-K. Chen, "Performance and Energy Implications of Many-Core Caches for Throughput Computing," *IEEE Micro*, vol. 30, no. 6, pp. 25–35, November 2010.

[12] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, "Power7: IBM's Next-Generation Server Processor," *IEEE Micro*, vol. 30, no. 2, pp. 7–15, Mar/Apr 2010.

[13] J. Shin, D. Huang, B. Petrick, C. Hwang, K. Tam, A. Smith, H. Pham, H. Li, T. Johnson, F. Schumacher, A. Leon, and A. Strong, "A 40 nm 16-Core 128-Thread SPARC SoC Processor," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 131–144, January 2011.

[14] Z. Guz, O. Itzhak, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser, "Threads vs. Caches: Modeling the Behavior of Parallel Workloads," in *International Conference on Computer Design (ICCD)*, October 2010, pp. 274–281.

[15] V. Narasiman, C. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *International Symposium on Microarchitecture (MICRO)*, December 2011, pp. 308–317.

[16] M. Gebhart, D. Johnson, D. Tarjan, S. W. Keckler, W. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *International Symposium on Computer Architecture (ISCA)*, 2011, pp. 235–246.

[17] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 272–283.

[18] NVIDIA Corporation, "CUDA C/C++ SDK CODE Samples," 2011.

[19] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-level Language Targeted to GPU Codes," in *Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[20] "Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php.

[21] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "A MapReduce Framework on Graphics Processors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2008, pp. 260–269.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[23] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *International Symposium on Workload Characterization (IISWC)*, November 2012, pp. 141–151.

[24] J. Mohd-Yusof, S. Swaminarayan, and T. C. Germann, "Co-Design for Molecular Dynamics: An Exascale Proxy Application," 2013. [Online]. Available: http://www.lanl.gov/orgs/adtsc/publications/science_highlights_2013/docs/Pg88_89.pdf

[25] "GPGPU-Sim," http://www.gpgpu-sim.org.

[26] NVIDIA Corporation, "NVIDIA CUDA Programming Guide," 2011.

[27] ——, "PTX: Parallel Thread Execution ISA Version 2.3," 2011.

[28] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[29] "GPGPU-Sim Manual," http://www.gpgpu-sim.org/manual.

[30] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating Cache Conflict Misses Through XOR-based Placement Functions," in *International Symposium on Supercomputing (ICS)*, July 1997, pp. 76–83.

[31] AMD Corporation, "ATI Stream Computing OpenCL Programming Guide," August 2010.

[32] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in *International Symposium on Microarchitecture (MICRO)*, December 2013, pp. 86–98.