DOI:10.1145/2682583

Learning Your Limit: Managing Massively Multithreaded Caches Through Scheduling

By Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt

Abstract

The gap between processor and memory performance has become a focal point for microprocessor research and development over the past three decades. Modern architectures use two orthogonal approaches to help alleviate this issue: (1) Almost every microprocessor includes some form of on-chip storage, usually in the form of caches, to decrease memory latency and make more effective use of limited memory bandwidth. (2) Massively multithreaded architectures, such as graphics processing units (GPUs), attempt to hide the high latency to memory by rapidly switching between many threads directly in hardware. This paper explores the intersection of these two techniques. We study the effect of accelerating highly parallel workloads with significant locality on a massively multithreaded GPU. We observe that the memory access stream seen by on-chip caches is the direct result of decisions made by the hardware thread scheduler. Our work proposes a hardware scheduling technique that reacts to feedback from the memory system to create a more cache-friendly access stream. We evaluate our technique using simulations and show a significant performance improvement over previously proposed scheduling mechanisms. We demonstrate the effectiveness of scheduling as a cache management technique by comparing cache hit rate using our scheduler and an LRU replacement policy against other scheduling techniques using an optimal cache replacement policy.

1. INTRODUCTION

Have you ever tried to do so many things that you could not get anything done? There is a classic psychological principal, known as unitary-resource theory, which states that the amount of attention humans can devote to concurrently performed tasks is limited in capacity.[16] Attempting to divide this attention among too many tasks at once can have a detrimental impact on your performance. We explore a similar issue in the context of highly multithreaded hardware and use human multitasking as an analogy to help explain our findings. Our paper studies architectures which are built to efficiently switch between tens of tasks on a cycle-by-cycle basis. Our paper asks the question: Even though massively multithreaded processors *can* switch between these tasks quickly, when *should* they?

Over the past 30 years, there has been a significant amount of research and development devoted to using on-chip caches more effectively. At a hardware level, cache management has typically been optimized by improvements to the hierarchy,[3] replacement/insertion policy,[14] coherence protocol,[19] or some combination of these. Previous work on hardware caching assumes that the access stream seen by the memory system is fixed. However, massively multithreaded systems introduce another dimension to the problem. Every clock cycle, a hardware thread scheduler must choose which of a core's active threads issues next. This decision has a significant impact on the access stream seen by the first level caches. In a massively multithreaded system, there are often many threads ready to be scheduled on each cycle. This paper exploits this observation and uses the thread scheduler to explicitly manage the access stream seen by the memory system to maximize throughput.

The primary contribution of this work is a *Cache-Conscious Wavefront Scheduling* (CCWS) system that uses locality information from the memory system to *shape* future memory accesses through hardware thread scheduling. Like traditional attempts to optimize cache replacement and insertion policies, CCWS attempts to predict when cache lines will be reused. However, cache way-management policies' decisions are made among a small set of blocks. A thread scheduler effectively chooses which blocks get inserted into the cache from a pool of potential memory accesses that can be much larger than the cache's associativity. Similar to how cache replacement policies effectively *predict* each line's re-reference interval,[14] our proposed scheduler attempts to *change* the re-reference interval to reduce the number of interfering references between repeated accesses to high locality data. Like cache tiling techniques performed in software by the programmer or compiler[25] that reduce the cache footprint of inner-loops by restructuring the code, CCWS reduces the aggregate cache footprint primarily by

This work was performed when Mike O'Connor was with Advanced Micro Devices (AMD) Research.

The original version of this paper is entitled "Cache-Conscious Wavefront Scheduling" and was published in the *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture*, 2012. Another version of this paper was also published as "Cache-Conscious Thread Scheduling for Massively Multithreaded Processors" in *IEEE Micro Special Issue: Micro's Top Picks from 2012 Computer Architecture Conferences* (2013).

dynamically throttling the number of threads sharing the cache in hardware.

The importance of fine-grained thread scheduling on cache management is a concern to any architecture where many hardware threads can share a cache. Some examples include Intel's Knights Corner,[11] Orcale's SPARC T4,[24] IBM's Blue Gene/Q,[9] and massively multithreaded *graphics processing units* (GPUs) such as those produced by AMD and NVIDIA. In this work, we study the effect of thread scheduling on GPUs since they represent the most extreme example of a many-thread architecture. However, our techniques could be extended and applied to any design where many threads share a cache.

1.1. The effect of increasing your capacity

One solution to increase multitasking capability is to simply increase the amount of information you can pay attention to at once. Increasing the size of the GPU's cache is one option our full paper[22] explores in more detail. However, no matter what size of cache is used, it will always have a limit and our work is focused on maximizing performance based on this limitation. Figure 1 highlights the potential benefit of increasing the attention capacity of a GPU. Figure 1 shows the speedup attained by increasing the *level one data* (L1D) cache size by 256× (from 32KB to 8MB) on the economically important server applications listed in Table 1. Our full paper[22] describes these applications in more detail. All of these applications see a 3× or more performance improvement with a much larger L1 data cache, indicating there is significant locality and that these programs and they would benefit from caching more information.

Given that our workloads have significant locality, the next question is where does it come from? To understand this, first we must explain the concept of a wavefront (or warp). A wavefront is a collection of threads whose instructions are executed in lock-step by a GPU. Each static assembly instruction is implicitly executed across multiple lanes when it is run. The number of threads in a wavefront is a value native to each machine. Our baseline hardware architecture has a wavefront size of 32. A consequence of this is that a single load/store instruction can access as many as 32 different cache lines when executed. We study the locality of our highly cache-sensitive benchmarks in Figure 2 which presents the average number of hits and misses *per thousand instructions* (PKI) using an unbounded L1D cache. The figure separates hits into two classes. We classify locality that occurs when data is initially referenced and re-referenced from the same wavefront as *intra-wavefront locality*. Locality resulting from data that is initially referenced by one wavefront and re-referenced by another is classified as *inter-wavefront locality*. Figure 2 illustrates that the majority of data reuse observed in our highly cache-sensitive benchmarks comes from intra-wavefront locality. If we were to think about this result in the context of a human trying to multitask, it means that each of your tasks require a large amount of information that is not shared with other tasks.

Figure 1. Performance using a round-robin scheduler at various L1D cache sizes for highly cache-sensitive workloads, normalized to a cache size of 32K. All caches are 8-way set-associative with 128B cache lines.

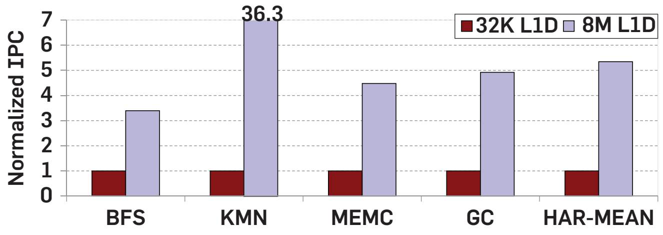


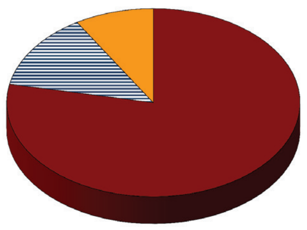
Table 1. GPU compute benchmarks (CUDA and OpenCL)

Highly cache sensitive			
Name	**Abbreviation**	**Name**	**Abbreviation**
BFS graph traversal[5]	BFS	Kmeans[5]	KMN
Memcached[10]	MEMC	Garbage collection[2]	GC

1.2. Primary goals of CCWS

Based on the observation that tasks do not share much data, we design CCWS with the principal goal of capturing more intra-wavefront locality. To this end, CCWS acts as a dynamic thread throttling mechanism that prevents wavefronts from issuing memory instructions when their accesses are predicted to interfere with intra-wavefront locality already present in the cache. CCWS detects if threads require more exclusive access to cache through memory system feedback from a lost locality detector (G in Figure 6 which gives a high-level view of our baseline architecture explained in more detail in Section 2). The lost locality detector observes when the L1D has been oversubscribed by storing the L1D's replacement victim tags in wavefront private sections. It uses these tags to determine if misses in the L1D could have been avoided by giving the wavefront that missed more exclusive cache access. The lost locality detector provides feedback akin to remembering that you were supposed to do a small errand but no longer remember what the errand was. You can use this information to reduce the number of tasks you are juggling at once because you realize you have exceeded the capacity of your attention. Interpreting this feedback is performed by CCWS's locality scoring system (H). The locality scoring system decides which wavefronts should be permitted to issue memory instructions, throttling those

Figure 2. Average hits and misses *per thousand instructions* (PKI) using an unbounded L1 data cache (with 128B lines) on highly cache-sensitive benchmarks.



■ Intra-wavefront hits PKI ▤ Inter-wavefront hits PKI ■ Misses PKI that are predicted to cause interference. Changing the thread scheduler to improve cache effectiveness introduces a new challenge not encountered by traditional cache management techniques. The goal of the scoring system is to maximize throughput, not just cache hit rate. Typically, massively multi-threaded architectures hide long latency operations by issuing instructions from more threads. The CCWS scoring system balances the trade-off between increasing latency tolerance and reducing the need for latency hiding. Increasing the number of actively interleaved threads improves latency tolerance, while interleaving less threads decreases the frequency of long latency events. This is like trying to find the right balance between not doing too many things at once while still managing to do more than one thing at a time.

1.3. The effect of intelligently managing your attention

Intelligently managing how your attention is allocated can increase your multitasking effectiveness. In the context of our problem, this is equivalent to improving the cache's replacement policy. To contrast the effects of hardware thread scheduling and replacement policy, consider Figures 3 and 4. They present the memory accesses created by two different thread schedulers in terms of cache lines touched. In this example, we assume each instruction generates four memory requests and we are using a fully associative, four-entry cache with a *least recently used* (LRU) replacement policy. Figure 3 illustrates what happens when the scheduler attempts to fill empty cycles by executing another wavefront. In Figure 3, no wavefront will find its data in cache because the data will have been evicted by requests from other wavefronts before the wavefront is scheduled again. In fact, even if a Belady optimal replacement policy[4] were used, only 4 hits in the cache are possible. The scheduler in Figure 4 is aware of the data reuse present in each wavefront's access stream and has rearranged the wavefront's issue order to be more cache-friendly. The scheduler in Figure 4 chooses to schedule the memory accesses from wavefront 0 together, even if it means leaving the processor idle while the requests for wavefront 0 return. This results in 12 cache hits, capturing every redundant access made by the wavefronts.

Figure 3. Example access pattern (represented as cache lines accessed) resulting from a throughput-oriented round-robin scheduler. The letters (A,B,C,...) represent cache lines accessed. W_i indicates which wavefront generated this set of accesses. For example, the first four accesses to cache lines A,B,C,D are generated by one instruction in wavefront 0.

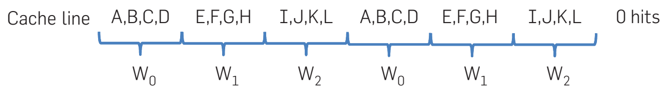
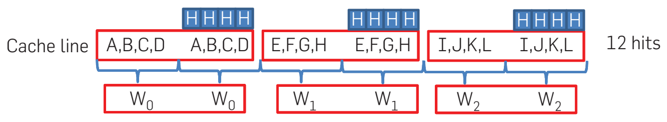


Figure 4. Example access pattern resulting from a hardware scheduler aware of its effect on the caching system. The red boxes highlight where scheduling improves locality by changing memory access order.



1.4. The effect of doing less all the time

Another simple way to decrease interference among tasks is to place a hard limit on the number of tasks performed at once. Our paper proposes a simple mechanism called *Static Wavefront Limiting* (SWL) that does something similar. SWL is implemented as a minor extension to the wavefront scheduling logic where a cap is placed on the number of wavefronts that are actively scheduled when the program is launched. Figure 5 shows the effect that limiting the number of wavefronts actively scheduled on a core has on the cache performance and system throughput of our highly cache-sensitive applications.

Figure 5 shows that peak throughput occurs at a multithreading value less than maximum concurrency, but greater than the optimum for cache miss rate (which limits concurrency to a single wavefront). In SWL, the programmer must specify a limit on the number of wavefronts when launching the kernel. This technique is useful if the user knows or can easily compute the optimal number of wavefronts prior to launching the kernel.

Our full paper[22] demonstrates that the optimal number of wavefronts is different for different benchmarks. Moreover, we find this number changes in each benchmark when its input data is changed. This dependence on benchmark and input data makes an adaptive CCWS system that adapts to locality in threads as they are run desirable.

2. GPU ARCHITECTURE

Our work studies modifications to the GPU-like accelerator architecture illustrated in Figure 6. The workloads we study are written in OpenCL or CUDA. Initially, an application begins execution on a host CPU which launches a kernel containing a large number of threads on the GPU. Our baseline system uses GPGPU-Sim 3.x.[1]

Our work focuses on the decision made by the *wavefront issue arbiter* (WIA) (Ⓐ in Figure 6). An in-order scoreboard (Ⓑ) and decode unit (Ⓒ) control when each instruction in an instruction buffer (I-Buffer Ⓓ) is ready to issue. The WIA decides which of these ready instructions issues next.

Figure 5. Average *misses per thousand instructions* (MPKI) and *harmonic mean* (HMEAN) performance improvement of highly cache-sensitive benchmarks with different levels of multithreading. *Instructions per cycle* (IPC) is normalized to 32 wavefronts.

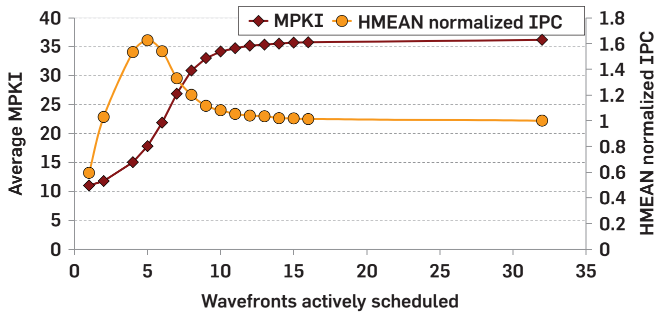
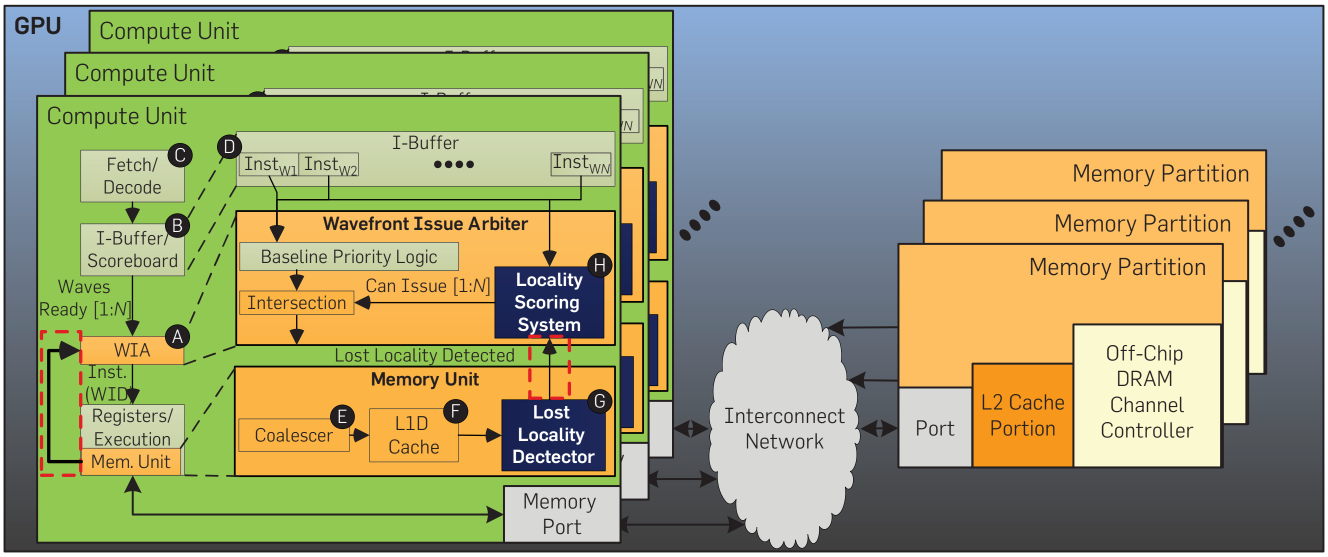


Figure 6. Overview of our GPU-like baseline accelerator architecture. $Inst_{wi}$ denotes the next instruction ready to issue for wavefront *i*. *N* is the maximum number of wavefront contexts stored on a core. I-Buffer: *instruction buffer*.



As mentioned previously, each memory instruction can generate more than one memory access. Modern GPUs attempt to reduce the number of memory accesses generated from each wavefront using an access coalescer (E) which groups the lane's memory requests into cache line-sized chunks when there is spatial locality across the wavefront. Applications with highly regular access patterns may generate as few as one or two memory requests that service all 32 lanes. Our baseline GPU includes a 32K L1 data cache (F) which receives memory requests from the coalescer.

3. CACHE-CONSCIOUS WAVEFRONT SCHEDULING (CCWS)

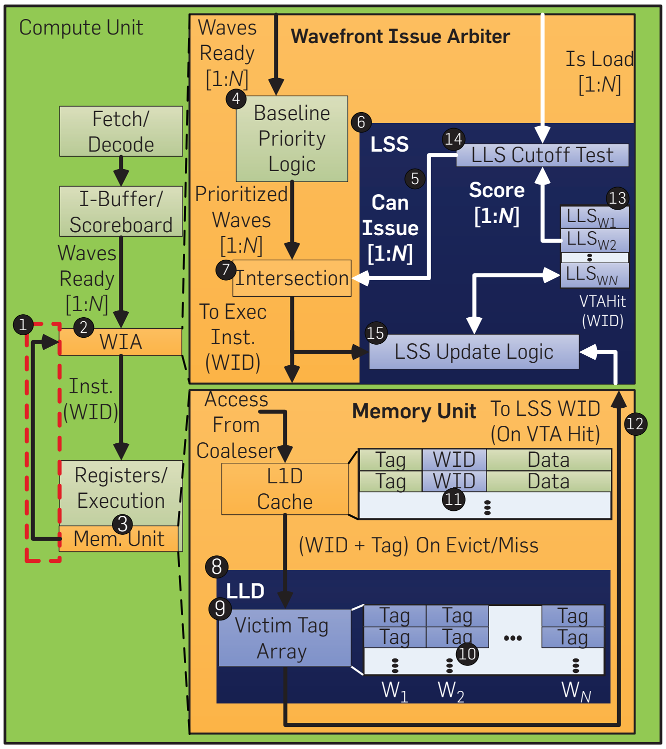
The goal of CCWS is to dynamically determine the number of wavefronts allowed to access the memory system and which wavefronts those should be. Figure 7 presents a more detailed view of CCWS's microarchitecture. At a high level, CCWS is a wavefront scheduler that reacts to access level feedback (1 in Figure 7) from the L1D cache and a *victim tag array* (VTA) at the memory stage.

CCWS's scheduling decisions are made by the locality scoring system (6). The intuition behind why the scoring system works is illustrated in Figure 8. Each wavefront is given a score based on how much *intra-wavefront locality* it has lost. These scores change over time. Wavefronts with the largest scores fall to the bottom of a small sorted stack (e.g., W_2 at T_1), pushing wavefronts with smaller scores above a cutoff (W_3 at T_1) which prevents them from accessing the L1D. In effect, the locality scoring system reduces the number of accesses between data re-references from the same wavefront by removing the accesses of other wavefronts.

3.1. Effect on baseline issue logic

Figure 7 shows the modifications to the baseline WIA (2) and memory unit (3) required for CCWS. CCWS is implemented as an extension to the system's baseline wavefront

Figure 7. Modeled GPU core microarchitecture. *N* is the number of wavefront contexts stored on a core. LSS: *locality scoring system*, LLD: *lost intra-wavefront locality detector*, WID: *wavefront ID*, LLS: lost-locality score, VTA: *victim tag array*, I-Buffer: *instruction buffer*.



prioritization logic (4). This prioritization could be done in a greedy, round-robin, or two-level manner. CCWS operates by preventing loads that are predicted to interfere with intra-wavefront locality from issuing through a *Can Issue* bit vector (5) output by the locality scoring system (6).

The intersection logic block (7) selects the highest priority ready wavefront that has issue permission.

3.2. Lost intra-wavefront locality detector (LLD)

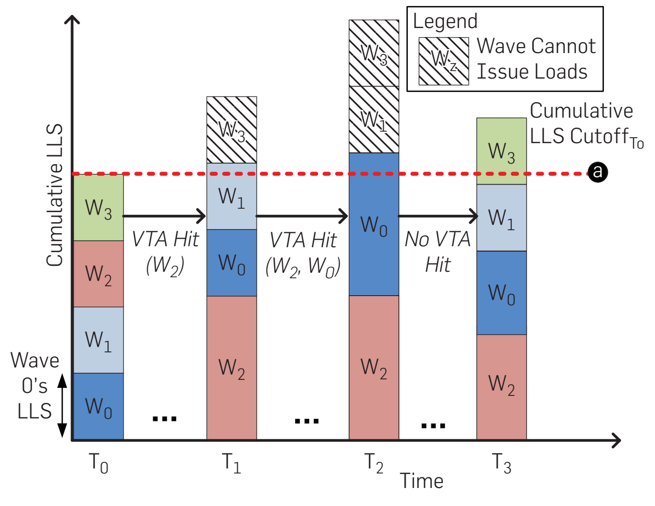
To evaluate which wavefronts are losing intra-wavefront locality, we introduce the LLD unit (8) which uses a *victim tag array* (VTA) (9). The VTA is a highly modified variation of a victim cache.[15] The entries of the VTA are subdivided among all the wavefront contexts supported on this core. This gives each wavefront its own small VTA (10). The VTA only stores cache tags and does not store line data. When a miss occurs and a line is reserved in the L1D cache, the *wavefront ID* (WID) of the wavefront reserving that line is written in addition to the tag (11). When that line is evicted from the cache, its tag information is written to that wavefront's portion of the VTA. Whenever there is a miss in the L1D cache, the VTA is probed. If the tag is found in that wavefront's portion of the VTA, the LLD sends a VTA hit signal to the locality scoring system (12). These signals inform the scoring system that a wavefront has missed on a cache line that may have been a hit if that wavefront had more exclusive access to the L1D cache.

3.3. Locality scoring system operation

Returning to the example in Figure 8, there are four wavefronts initially assigned to the compute unit. Time T_0 corresponds to the time these wavefronts are initially assigned to this core. Each segment of the stacked bar represents a score given to each wavefront to quantify the amount of intra-wavefront locality it has lost (which is related to the amount of cache space it requires). We call these values *lost-locality scores* (LLS). At T_0 we assign each wavefront a constant base locality score. LLS values are stored in a max heap (13) inside the locality scoring system. A wavefront's LLS can increase when the LLD sends a VTA hit signal for this wavefront. The scores each decrease by one point every cycle until they reach the base locality score. The locality scoring system gives wavefronts losing the most intra-wavefront locality more exclusive L1D cache access by preventing the wavefronts with the smallest LLS from issuing load instructions. Wavefronts whose LLS rise above the cumulative LLS cutoff (a in Figure 8) in the sorted heap are prevented from issuing loads.

The LLS cutoff test block (14) takes in a bit vector from the instruction buffer, indicating what wavefronts are attempting to issue loads. It also takes in a sorted list of LLSs, performs a prefix sum, and clears the *Can Issue* bit for wavefronts attempting to issue loads whose LLS is above the cutoff. In our example from Figure 8, between T_0 and T_1, W_2 has received a VTA hit and its score has been increased. W_2's higher score has pushed W_3 above the cumulative LLS cutoff, clearing W_3's *Can Issue* bit if it attempts to issue a load instruction. From a microarchitecture perspective, LLSs are modified by the score update logic (15). The update logic block receives VTA hit signals (with a WID) from the LLD which triggers a change to that wavefront's LLS. In Figure 8, between T_1 and T_2 both W_2 and W_0 have received VTA hits, pushing both W_3 and W_1 above the cutoff. Between T_2 and T_3, no VTA hits have occurred and the scores for W_2 and W_0 have decreased enough to allow both W_1 and W_3 to issue loads again. This illustrates how the system naturally backs off thread throttling over time. Our paper explains the scoring system in more detail.

Figure 8. Locality scoring system operation example. LLS: *lost-locality score.*



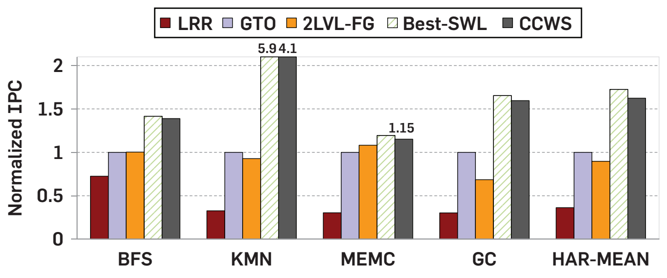
4. EVALUATION

We model CCWS in GPGPU-Sim[1] (version 3.1.0). The simulator is configured to model an NVIDIA Quadro FX5800, extended with L1 data caches and an L2 unified cache similar to NVIDIA Fermi. We also evaluate L1 data cache hit rate using the Belady-optimal replacement policy,[4] which chooses the line which is re-referenced furthest in the future for eviction. Belady replacement is evaluated using a trace-based cache simulator that takes GPGPU-Sim cache access traces as input. We ran our experiments on the highly cache-sensitive applications listed in Table 1. The full version of our paper[22] provides additional data on a collection of moderately cache-sensitive and cache-insensitive workloads and gives more details on our experimental setup.

The data in Figures 9 and 10 is collected using GPGPU-Sim for the following mechanisms:

LRR: Loose round-robin scheduling. Wavefronts are prioritized for scheduling in round-robin order. However, if a wavefront cannot issue during its turn, the next wavefront in round-robin order is given the chance to issue.

Figure 9. Performance of various schedulers for the highly cache-sensitive benchmarks. Normalized to the GTO scheduler.



GTO: A greedy-then-oldest scheduler. GTO runs a single wavefront until it stalls then picks the oldest ready wavefront.

2LVL-GTO: A two-level scheduler similar to that described by Narasiman et al.[20] Their scheme subdivides wavefronts waiting to be scheduled on a core into *fetch groups* (FG) and executes from only one fetch group until all wavefronts in that group are stalled. Intra- and inter-FG arbitration is done in a GTO manner.

Best-SWL: An oracle solution that knows the optimal number of wavefronts to schedule concurrently before the kernel begins to execute.

CCWS: Cache-Conscious Wavefront Scheduling with GTO wavefront prioritization logic.

The data for Belady-optimal replacement *misses per thousand instructions* (MPKI) presented in Figure 10 is generated with our trace-based cache simulator:

(scheduler)-BEL: Miss rate reported by our cache simulator when using the Belady-optimal replacement policy. The access streams generated by running GPGPU-Sim with the specified (scheduler) are used.

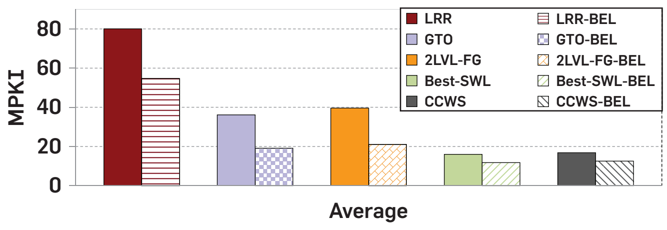
4.1. Performance

Figure 9 shows that CCWS achieves a harmonic mean 63% performance improvement over a simple greedy wavefront scheduler and 72% over the 2LVL-GTO scheduler on highly cache-sensitive benchmarks. The GTO scheduler performs well because prioritizing older wavefronts allows them to capture intra-wavefront locality by giving them more exclusive access to the L1 data cache.

CCWS and SWL provide further benefit over the GTO scheduler because these programs have a number of uncoalesced loads, touching many cache lines in relatively few memory instructions. Therefore, even restricting to just the oldest wavefronts still touches too much data to be contained by the L1D.

Figure 10 shows the average MPKI for all of our highly cache-sensitive applications using an LRU replacement policy and an oracle Belady-optimal replacement policy (BEL). It illustrates that the reason for the performance advantage provided by the wavefront limiting schemes is a sharp decline in the number of L1D misses. Furthermore, it demonstrates that a poor choice of scheduler can reduce the effectiveness of any replacement policy.

Figure 10. MPKI of various schedulers and replacement policies for the highly cache-sensitive benchmarks.



4.2. Detailed breakdown of wavefront locality

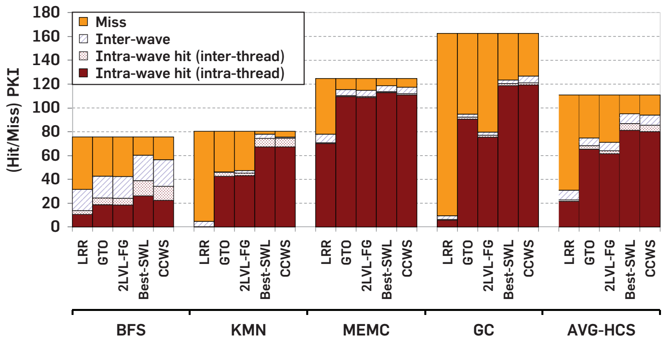
Figure 11 breaks down L1D accesses into misses, inter-wavefront hits, and intra-wavefront hits for our evaluated the schedulers. In addition, it quantifies the portion of intra-wavefront hits that are a result of intra-thread locality. It illustrates that the decrease in cache misses using CCWS and Best-SWL comes chiefly from an increase in intra-wavefront hits. Moreover, the bulk of these hits are a result of intra-thread locality. The exception to this rule is *Breadth First Search* (BFS) graph traversal, where only 30% of intra-wavefront hits come from inter-thread locality and we see a 23% increase in inter-wavefront hits. An inspection of the code reveals that inter-thread sharing (which manifests itself as both intra-wavefront and inter-wavefront locality) occurs when nodes in the application's input graph share neighbors. Limiting the number of wavefronts actively scheduled increases the hit rate of these accesses because it limits the amount of nonshared data in the cache, increasing the chance that these shared accesses hit.

5. RELATED WORK

Other papers have observed that throttling the number of threads running in a system can improve performance. Bakhoda et al.[1] observe that launching less workgroups (or CTAs) on a GPU core without an L1 cache improved performance by alleviating contention for the memory system. Guz et al.[8] introduce an analytical model to quantify the *performance valley* that exists when the number of threads sharing a cache is increased. They show that increasing the thread count increases performance until the aggregate working set no longer fits in cache. Increasing threads beyond this point degrades performance until enough threads are present to hide the system's memory latency. In effect, CCWS dynamically detects when a workload has entered the machine's *performance valley* and scales down the number of threads sharing the cache to compensate. Cheng et al.[6] introduce a thread throttling scheme to reduce memory latency in multithreaded CPU systems. They propose an analytical model and memory task limit throttling mechanism to limit thread interference in the memory stage.

There is a body of work attempting to increase cache

Figure 11. Breakdown of L1D misses, intra-wavefront locality hits (broken into intra-thread and inter-thread), and inter-wavefront locality hits per thousand instructions for highly cache-sensitive benchmarks.



hit rate by improving the replacement policy (e.g., Jaleel et al.[14] among many others). All these attempt to exploit different heuristics of program behavior to predict a block's re-reference interval and mirror the Belady-optimal[4] policy as closely as possible. While CCWS also attempts to maximize cache efficiency, it does so by shortening the re-reference interval rather than by predicting it. CCWS has to balance the shortening of the re-reference interval by limiting the number of eligible wavefronts while still maintaining sufficient multithreading to cover most of the memory and operation latencies. Other schemes attempt to manage interference among heterogeneous workloads,[12, 21] but each thread in our workload has roughly similar characteristics. Recent work has explored the use of prefetching on GPUs.[18] However, prefetching cannot improve performance when an application is bandwidth limited whereas CCWS can help in such cases by reducing off-chip traffic. Concurrent to our work, Jaleel et al.[13] propose the CRUISE scheme which uses LLC utility information to make high-level scheduling decisions in multiprogrammed chip multiprocessors (CMPs). Our work focuses on the first level cache in a massively multithreaded environment and is applied at a much finer grain. Scheduling decisions made by CRUISE tie programs to cores, where CCWS makes issue-level decisions on which bundle of threads should enter the execution pipeline next.

Others have also studied issue-level scheduling algorithms on GPUs. Lakshminarayana and Kim[17] explore numerous warp scheduling policies in the context of a GPU without hardware-managed caches and show that, for applications that execute symmetric (balanced) dynamic instruction counts per warp, a fairness-based warp and DRAM access scheduling policy improves performance. Several works have explored the effect of two-level scheduling on GPUs.[7, 20] A two-level scheduler exploits inter-wavefront locality while ensuring wavefronts reach long latency operations at different times by scheduling groups of wavefronts together. However, Figure 2 demonstrates that the highly cache--sensitive benchmarks we studied will benefit more from exploiting intra-wavefront locality than inter-wavefront locality. Previous schedulers do not take into account the effect issuing more wavefronts has on the intra-wavefront locality of those wavefronts that were previously scheduled. In the face of L1D thrashing, the round-robin nature of their techniques will cause the destruction of older wavefront's intra-wavefront locality. Our follow-up work on *Divergence-Aware Warp Scheduling* (DAWS)[23] builds upon the insights in CCWS. DAWS attempts to proactively predict the cache footprint of each warp. It does this by considering the impact of memory and control divergence while taking into account the loop structure of computation kernels. In that work, we demonstrate through an example that DAWS enables non-optimized GPU code, where locality is not managed by the programmer, to perform within 4% of optimized code.

6. CONCLUSION

Current GPU architectures are excellent at accelerating applications with copious parallelism, whose memory access is regular and statically predicable. Modern CPUs feature deep cache hierarchies and a relatively large amount of cache available per-thread, making them better suited for workloads with irregular locality. However, CPU's limited thread count and available memory bandwidth prohibit their ability to exploit pervasive parallelism. Each design has problems running the important class of highly parallel irregular applications where threads access data from disparate regions of memory. The question of how future architectures can accelerate these applications is important.

Many highly parallel, irregular applications are commercially important and found in modern data centers. The highly cache-sensitive benchmarks in our paper include several such workloads, including Memcached,[10] a parallel Garbage Collector,[2] and a breadth-first search graph traversal[5] program. We demonstrate that these applications are highly sensitive to L1 cache capacity when naive thread schedulers are used. Furthermore, we show that a relatively small L1 cache can capture their locality and improve performance, provided an intelligent issue-level thread scheduling scheme is used.

Although our work focuses primarily on performance, the impact of CCWS and consequently fine-grained thread scheduling on power consumption is important. As modern chips become progressively more power limited, preserving locality in the cache can be an effective way to reduce power consumption. CCWS can be tuned to reduce the number of data cache misses even further, at the expense of some performance.

Intellectually, we feel this work offers a new perspective on fine-grained memory system management. We believe that integrating the cache system with the issue-level thread scheduler to change the access stream seen by the memory system opens up a new direction of research in cache management. CCWS demonstrates that a relatively simple, dynamically adaptive, feedback-driven scheduling system can vastly improve the performance of an important class of applications.

Acknowledgments

We thank Norm Jouppi, Yale N. Patt, Aamer Jaleel, Wilson Fung, Hadi Jooybar, Inderpreet Singh, Tayler Hetherington, Ali Bakhoda, and our anonymous reviewers for their insightful feedback. We also thank Rimon Tadros for his work on the garbage collector benchmark. This research was funded in part by a grant from Advanced Micro Devices Inc. C

References

1. Bakhoda, A., Yuan, G., Fung, W., Wong, H., Aamodt, T. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, 163–174.
2. Barabash, K., Petrank, E. Tracing garbage collection on highly parallel platforms. In *Proceedings of International Symposium on Memory Management (ISMM 2010)*, 1–10.
3. Beckmann, B.M., Marty, M.R., Wood, D.A. ASR: Adaptive selective replication for CMP caches. In *Proceedings of International Symposium on Microarchitecture (MICRO 39)* (2006), 443–454.
4. Belady, L.A. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J. 5*, 2 (1966), 78–101.
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K. Rodinia: A benchmark suite

for heterogeneous computing. In *Proceedings of International Symposium on Workload Characterization (IISWC 2009)*, 44–54.
6. Cheng, H.Y., Lin, C.H., Li, J., Yang, C.L. Memory latency reduction via thread throttling. In *Proceedings of International Symposium on Microarchitecture (MICRO 43)* (2010), 53–64.
7. Fung, W., Aamodt, T. Thread block compaction for efficient SIMT control flow. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA 2011)*, 25–36.
8. Guz, Z., Bolotin, E., Keidar, I., Kolodny, A., Mendelson, A., Weiser, U. Many-core vs. many-thread machines: Stay away from the valley. *Comput. Architect. Lett. 8*, 1 (Jan. 2009), 25–28.
9. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G.T., Boyle, P., Chist, N., Kim, C. The IBM Blue Gene/Q compute chip. *Micro IEEE 32*, 2 (Mar.–Apr. 2012), 48–60.
10. Hetherington, T.H., Rogers, T.G., Hsu, L., O'Connor, M., Aamodt, T.M. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS 2012)*, 88–98.
11. Intel Xeon Phi Coprocessor Brief. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.
12. Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, S., Jr., Emer, J. Adaptive insertion policies for managing shared caches. In *Proceedings of International Conference on Parallel Architecture and Compiler Techniques (PACT 2008)*, 208–219.
13. Jaleel, A., Najaf-abadi, H.H., Subramaniam, S., Steely, S.C., Emer, J. CRUISE: Cache replacement and utility-aware scheduling. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, 249–260.
14. Jaleel, A., Theobald, K.B., Steely, S.C., Jr., Emer, J. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of International Symposium on Computer Architecture (ISCA 2010)*, 60–71.
15. Jouppi, N.P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of International Symposium on Computer Architecture (ISCA 1990)*, 364–373.
16. Kahneman D. Attention and Effort Prentice-Hall, 1973.
17. Lakshminarayana, N.B., Kim, H. Effect of instruction fetch and memory scheduling on GPU performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU* (2010).
18. Lee, J., Lakshminarayana, N.B., Kim, H., Vuduc, R. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proceedings of International Symposium on Microarchitecture (MICRO 43)* (2010), 213–224.
19. Marty, M.R., Hill, M.D. Coherence ordering for ring-based chip multiprocessors. In *Proceedings of International Symposium on Microarchitecture (MICRO 39)* (2006), 309–320.
20. Narasiman, V., Shebanow, M., Lee, C.J., Miftakhutdinov, R., Mutlu, O., Patt, Y.N. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of International Symposium on Microarchitecture (MICRO 44)* (2011), 308–317.
21. Qureshi, M.K., Patt, Y.N. Utility based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of International Symposium on Microarchitecture (MICRO 39)* (2006), 423–432.
22. Rogers, T.G., O'Connor, M., Aamodt, T.M. Cache-Conscious Wavefront Scheduling. In *Proceedings of IEEE/ACM International Symposiumon Microarchitecture (MICRO-45)* (2012).
23. Rogers, T.G., O'Connor, M., Aamodt, T.M. Divergence-Aware Warp Scheduling. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO-46)* (2013).
24. Shah, M., Golla, R., Grohoski, G., Jordan, P., Barreh, J., Brooks, J., Greenberg, M., Levinsky, G., Luttrell, M., Olson, C., Samoail, Z., Smittle, M., Ziaja, T. Sparc T4: A dynamically threaded server-on-a-chip. *Micro IEEE 32*, 2 (Mar.–Apr. 2012), 8–19.
25. Wolf, M.E., Lam, M.S. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)* (1991), 30–44.

Timothy G. Rogers and Tor M. Aamodt ({tgrogers, aamodt}@ece.ubc.ca), Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada.

Mike O'Connor (moconnor@nvidia.com), NVIDIA Research, Austin, TX.