Exploiting Core Criticality for Enhanced GPU Performance

Adwait Jog[1] Onur Kayiran[2] Ashutosh Pattnaik[3] Mahmut T. Kandemir[3]
Onur Mutlu[4,5] Ravishankar Iyer[6] Chita R. Das[3]

[1]College of William and Mary [2]Advanced Micro Devices, Inc.
[3]Pennsylvania State University [4]ETH Zürich [5]Carnegie Mellon University [6]Intel

ABSTRACT

Modern memory access schedulers employed in GPUs typically optimize for memory throughput. They implicitly assume that *all* requests from different cores are equally important. However, we show that during the execution of a subset of CUDA applications, different cores can have different amounts of tolerance to latency. In particular, cores with a larger fraction of warps waiting for data to come back from DRAM are less likely to tolerate the latency of an outstanding memory request. Requests from such cores are more critical than requests from others. Based on this observation, this paper introduces a new memory scheduler, called *(C)ritica(L)ity (A)ware (M)emory (S)cheduler* (CLAMS), which takes into account the latency-tolerance of the cores that generate memory requests. The key idea is to use the fraction of critical requests in the memory request buffer to switch between scheduling policies optimized for criticality and locality. If this fraction is below a threshold, CLAMS prioritizes critical requests to ensure cores that cannot tolerate latency are serviced faster. Otherwise, CLAMS optimizes for locality, anticipating that there are too many critical requests and prioritizing one over another would not significantly benefit performance.

We first present a core-criticality estimation mechanism for determining critical cores and requests, and then discuss issues related to finding a balance between criticality and locality in the memory scheduler. We progressively devise three variants of CLAMS, and show that the *Dynamic CLAMS* provides significantly higher performance, across a variety of workloads, than the commonly-employed GPU memory schedulers optimized solely for locality. The results indicate that a GPU memory system that considers both core criticality and DRAM access locality can provide significant improvement in performance.

1. INTRODUCTION

Graphics Processing Units (GPUs) are becoming increasingly popular for general purpose computing due to their capability in providing large improvements in performance and energy efficiency compared to CPUs [1–3, 11, 24, 49, 55–57, 59, 69, 77, 80, 90]. GPUs achieve significant speedups by exploiting high thread-level parallelism (TLP). They launch thousands of threads across multiple cores to mask the performance bottlenecks of a single thread. Threads are typically grouped into fixed-sized batches known as *warps* or *wavefronts*. To serve the high memory-demands of thousands of concurrently executing threads, many modern GPUs use GDDR5 memory, which typically has 3-4 times the peak-bandwidth of the high-end DDR4 memories used in chip multiprocessors.

Although high-bandwidth memory systems have increased GPU performance substantially, memory bandwidth is still precious and a critical performance determinant [27, 31, 39, 41, 43, 68, 72, 88]. In fact, it will be more so as compute resources on GPUs continue to increase [28, 38, 41, 43]. To address this, a large body of work has focused on improving bandwidth utilization (e.g., [4, 7, 40, 41, 68, 88]), and caching efficiency in GPUs (e.g., [5, 35, 41, 58, 75]). However, past works primarily focus on improving application performance by treating all threads and memory requests with *equal* importance. This phenomenon stems from the fact that GPUs typically focus on improving the collective performance of multiple concurrently executing threads, by overlapping their execution. In the same vein, the commonly-used memory request scheduling policy, First-ready First-Come First-Served (FR-FCFS) [73, 74, 93], implicitly assumes that all memory requests are *equally critical* for overall performance, and hence, it aims to maximize memory data throughput rather than minimize latency of specific requests or cores.

We observe that, in a subset of CUDA applications, due to the contention of memory requests from different cores in the memory system (along with interconnects and shared caches), and the inability of the FR-FCFS memory scheduler to distinguish between the memory requests originating from different cores, GPU cores experience significant variation in average memory access latencies. Due to such variation, the number of stalling warps that belong to the cores that suffer from higher memory access latencies is typically higher than that of other cores, making the former type of cores less latency tolerant, i.e., more *critical* for overall performance. Thus, because different cores have varying degrees of tolerance to latency during execution, their corresponding memory requests have varying degrees of criticality.

In contrast to the purely locality-focused memory schedulers, **our goal** in this work is to design a memory scheduler that is cognizant of the latency tolerance of cores. One simple idea based on our observation is to detect and *always prioritize* critical requests over non-critical requests.

SIGMETRICS '16, June 14-18, 2016, Antibes Juan-Les-Pins, France
© 2016 ACM. ISBN 978-1-4503-4266-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2896377.2901468

As the cores that lack enough warps to hide the long memory latencies are more likely to quickly stall for the data to come back, prioritizing requests from such cores in the memory controller provides a way of pro-actively avoiding them from getting stalled. However, we find that such a memory scheduler that is focused *purely* on core criticality degrades DRAM access locality significantly. This motivates us to explore more intelligent memory scheduling schemes that consider *both criticality and locality*. To this end, we introduce the *(C)ritica(L)ity (A)ware (M)emory (S)cheduler* (CLAMS) for GPUs, which achieves a fine balance between core criticality and DRAM access locality.

The CLAMS design comprises four steps. First, it periodically calculates the current level of latency tolerance of a GPU core. It does so by periodically calculating the fraction of short-latency warps on the core.[1] A core is expected to be more latency tolerant if most of the launched warps are short-latency warps that execute compute instructions or that find their required data in privates caches. Second, CLAMS periodically ranks the cores based on their current level of latency tolerance, and tags the memory requests with the core's rank. The ranking is done in such a way that the cores that have lower latency tolerance are ranked lower. Third, based on the value of the rank, CLAMS determines whether or not a request should be considered critical. To do so, it uses a criticality-rank threshold (Th_{CR}), which specifies *up to which rank* a request should be considered as *critical*. Fourth, CLAMS decides whether or not a critical request should be prioritized in the memory scheduler, by also taking into account DRAM access (row-buffer) locality. It does so by periodically calculating the percentage of requests that are considered as critical in the memory request buffer, and comparing it to a value called the scheduling-mode threshold (Th_{SM}). If the percentage of critical requests is below Th_{SM}, CLAMS goes into criticality mode, where it prioritizes critical requests to ensure cores that cannot tolerate latency are serviced faster. Otherwise, CLAMS operates in locality mode, where it optimizes for locality (like existing schedulers), anticipating that there are too many critical requests to prioritize one over another.

To our knowledge, this is the first work that observes latency tolerance differences between GPU cores and exploits such differences to improve GPU resource management, focusing on memory request scheduling. This paper makes the following **contributions:**

• We introduce the concept of core-criticality in GPUs. We show that all GPU cores do *not* possess the same latency tolerance at all times, and this variation in latency tolerance across cores is one of the key reasons for different levels of criticality among memory requests, which is not exploited by current GPU memory request schedulers.

• We introduce the first GPU memory scheduler, CLAMS, which takes into account core-criticality and achieves a fine balance between criticality and locality via our new dynamic criticality estimation mechanism. We propose three different designs for CLAMS: static, semi-dynamic (Semi-Dyn) and dynamic (Dyn) based on how required thresholds (Th_{CR} and Th_{SM}) are computed, and find that Dyn-CLAMS is the best performer due to its ability to compute these thresholds at runtime and thereby adapt dynamically to varying application demands.

[1]Section 3 describes our exact mechanism to measure the latency tolerance of a core.

• We present a comprehensive experimental evaluation of three CLAMS designs as compared to commonly used FR-FCFS [73, 74, 93] and FR-FCFS-Cap [62] memory schedulers using a variety of CUDA applications. Our results show that Dyn-CLAMS reduces the latency of critical memory requests by 35%, resulting in an average 9% IPC improvement (maximum 15%) over the FR-FCFS scheduler, and is within 1% of an idealized CLAMS design that uses best threshold values profiled separately for *each* application. Furthermore, the performance of none of the evaluated applications degrades with our mechanism.

2. BACKGROUND AND MOTIVATION

A typical GPU consists of multiple simple cores, also called streaming-multiprocessors[2] [67] in NVIDIA terminology. Each core is associated with private L1 data, texture and constant caches, along with software-managed scratchpad memory. The cores are connected to memory channels (partitions) via an interconnection network. Each memory partition is associated with a shared L2 cache, and its associated memory requests are handled by a GDDR5 memory controller. When an application kernel is launched on a GPU, the memory requests originating from different cores interfere at various levels of the GPU memory hierarchy, such as the interconnect, last-level caches, and main-memory. At each of these levels, the underlying shared resource management policies do *not* consider the source core-ids of the requests while making decisions, and therefore might allocate shared resources across different cores in an *uneven* fashion. Our detailed analysis shows that such uneven allocation can lead to significant variation in average memory latencies across different GPU cores.

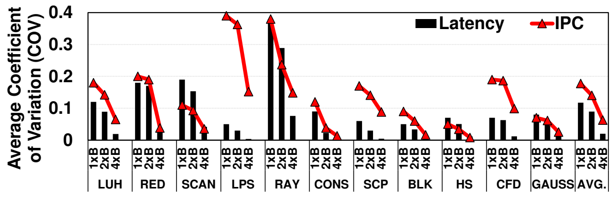


Figure 1: Average Coefficient of Variation (COV) in memory access latencies and in IPCs across different GPU cores.

To understand this variation, we calculate the *coefficient of variation (COV)* in memory access latencies, which is defined as the ratio of standard deviation over arithmetic mean of the average memory access latencies experienced by different cores. Figure 1 shows the COV in memory access latencies as well as COV in IPCs across different cores for 11 different CUDA applications, averaged across fixed epochs[3], for three different scenarios: applications when executed on a GPU that has 1) equal to (1xB), 2) double (2xB), and 3) quadruple (4xB) the peak memory bandwidth of our baseline GPU architecture. In the baseline scenario (1xB), we observe significant COV in latencies across cores for applications like `LUH` (13%), `RAY` (33%), `SCAN` (19%), and `RED` (18%). Because of such variation, many GPU cores experience higher memory access latencies than others. Therefore, such cores have a high number of stalling warps on memory,

[2]In this paper, we use the term *core* for a streaming-multiprocessor (SM).

[3]Epoch length is 10K cycles.

making them *less latency tolerant* than other cores. This might also result in higher stall times for these cores, leading to significant COV in IPCs across different cores, as shown in Figure 1.

We observe significant COV in both latencies and IPCs for RAY, LUH, and RED, stressing the fact that variation in latencies can lead to significant variation in IPCs across cores. In addition to this observation, we notice two contrasting cases. First, in SCAN, variation in latencies across cores is higher than variation in IPCs. This is because SCAN is able to tolerate higher latencies up to a certain level, which reduces the variation in IPCs. Second, in LPS and CFD, much lower variation in latencies across cores is present compared to variation in IPCs. This is because CTA and instruction-mix load imbalance [6,43,54,89] across cores also causes IPC variation during application execution. Finally, we observe that increasing the peak memory bandwidth leads to a significant decrease in COV in latencies as well as in IPCs, implying that memory bandwidth contention of different cores in the memory system is a major cause of COV in latencies as well as in IPCs.

Our goal is to develop a mechanism that prioritizes the cores that suffer from lower latency tolerance. Improving the performance of these cores would improve overall system performance by enabling these cores to make progress (Section 3.1). We expect such a mechanism to specifically benefit those applications that have significant COV in both latencies and IPCs (e.g., RAY, LUH, and RED), and not so for those applications that have small COV (e.g., BLK, HS). Such a mechanism can be employed at various levels. For example, a warp scheduler can be employed to control the progress of each core separately. However, since the cores contend for the memory system resources, especially main memory bandwidth, a memory scheduling mechanism can be more effective to expedite the requests of cores with lower latency tolerance, and is therefore the focus of this paper. Thus, in this work, we design a new GPU memory scheduler that is aware of the latency tolerance of individual cores.

3. CORE CRITICALITY: BASIC IDEAS AND METRICS

In this section, we describe and analyze the metrics to gauge the latency tolerance of a core and its variance across cores.

3.1 Latency Tolerance as a Measure of Core Criticality

A GPU core achieves high latency tolerance by hosting a large number of warps. If some warps get blocked because of pending DRAM accesses, the remaining warps can continue their execution and potentially mask the long latency penalties of the blocked warps. In order to quantify the level of latency tolerance on a GPU core, we employ a two-step strategy. The first step is to classify the warps as *short-latency* and *long-latency* warps. The short-latency warps are the issued warps that do not have any pending memory request(s). In other words, they are currently executing either compute instructions or instructions whose required data is already present in the private caches. Therefore, these short-latency warps are expected to provide latency tolerance to the core in the presence of the remaining long-latency warps whose execution might be blocked for hundreds of cycles due to pending memory request(s). After classifying the warps into short-latency vs. long-latency, the second step is to periodically[4] calculate *the ratio of short-latency warps over total issued warps.* Note that the sum of short-latency and long-latency warps is equal to the total issued warps, and thus this ratio can take a value between 0 and 1. *We use this ratio as a metric to gauge the latency tolerance of a core.* We call it the *short-latency ratio.* A high value of this ratio indicates that the core has a high percentage of short-latency warps, suggesting that the core has high latency tolerance.

We observe in our experiments that, relative change in IPC and relative change in our latency tolerance metric has an average correlation of 74% across our application suite (described in Section 6). This means that improving the latency tolerance of a core might improve its IPC. We also observe that a mechanism that prioritizes the memory requests of cores that have lower latency tolerance can have higher impact on overall performance. For example, it is more advantageous to make one more additional warp ready to execute in a core with zero short-latency warps (i.e., a core with no latency tolerance) compared to doing so in a core with a large number of short-latency warps (i.e., a core with enough latency tolerance).

To analyze the criticality of cores, we quantize our latency tolerance metric, short-latency ratio, into eight equal parts.[5] Based on this value, we assign a criticality *rank* to a core. Essentially, each equal part of the short-latency ratio corresponds to a rank. For example, if this ratio is less than or equal to $\frac{1}{8}$, the core is considered to be the most critical and is in *rank-1* state. Similarly, if a core's short-latency ratio is greater than $\frac{7}{8}$, that core is considered to be the least critical, and is in *rank-8* state.

Formally, we consider a core to be *critical* if the current *rank* of the core is less than or equal to a *Criticality-Rank-Threshold* (Th_{CR}). In other words, the value of Th_{CR} specifies ***up to which rank*** the core should be considered as *critical.* For example, a Th_{CR} value of 4 implies that the core is considered critical only if its current rank is less than or equal to rank 4.

3.2 Understanding Variation of Criticality Across Cores

Not only the latency tolerance of a core can change during execution, but we also observe that there is a *wide variation in latency tolerance across GPU cores.* To measure this variation, we introduce a metric, called *percentage of critical cores (PCC)*, which is defined as the percentage of GPU cores that are in the critical state. Since a core is treated as critical based on the chosen value of Criticality-Rank-Threshold (Th_{CR}), PCC needs to be defined for a particular value of Th_{CR}. Hence, we define PCC(Th_{CR}) as the percentage of critical cores (PCC) for a particular Th_{CR}, where Th_{CR} can take any integer from 1 to 8. If PCC(Th_{CR}) is 100%, it means that all cores are critical, and have similar latency tolerance. If PCC(Th_{CR}) is 0%, it means that all cores are non-critical. In both

[4]In our experiments, we calculate this ratio over an epoch of 128 cycles (a threshold that is determined empirically).

[5]We could divide this ratio into more than eight parts to get a finer granularity picture of the current latency tolerance of a core, but our detailed studies show that eight parts provide sufficient granularity to understand and distinguish between the latency tolerance of different cores.

cases, the variation in criticality across cores is insignificant. On the other hand, if the value of PCC(Th_{CR}) is in mid-range, then some cores are critical and the remaining cores are not. Therefore, the value of PCC(Th_{CR}) gives a notion of the variation in criticality across cores. For a better understanding of the PCC metric, consider Figure 2, which shows the rank of three cores of a GPU. We notice that if Th_{CR} is chosen as 1, PCC is 0%, as all cores have higher rank than 1 and no core is considered critical.

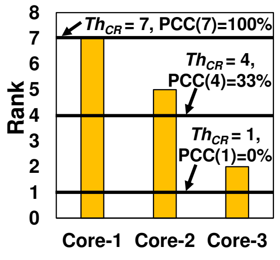


Figure 2: Example demonstrating the PCC metric.

With Th_{CR} equal to 4, the value of PCC is 33%, as the rank of core-3 is less than the chosen value of Th_{CR}, making it the only critical core in the system. With Th_{CR} equal to 7, the value of PCC is 100%, as all cores are considered critical because the rank of every core is less than Th_{CR}.

We observe that as Th_{CR} increases, the number of critical cores also increases (or remains the same). Therefore, PCC(Th_{CR}) also increases (or remains the same) as Th_{CR} increases from 1 to 8. Formally, $[PCC(a) \geq PCC(b)]$, if $(a > b)$.

3.2.1 Analysis of the PCC metric.

Figure 3 shows PCC(Th_{CR}) over time (sampled every 1K cycles) for four applications, for three different values of Th_{CR} (Th_{CR} = {1, 4, 7}), over time. We make three main observations from this figure.

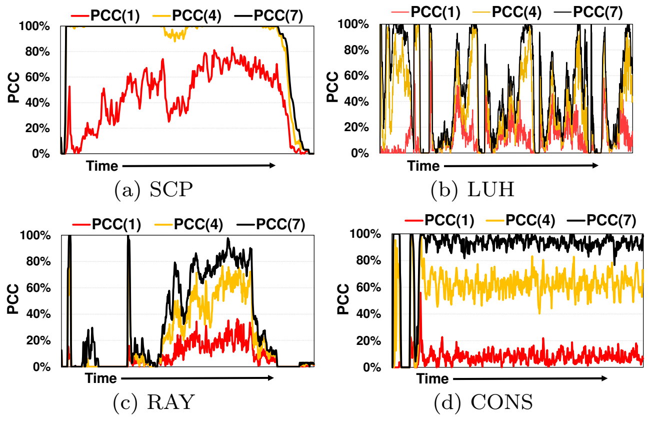


Figure 3: PCC metric over time with different criticality-rank thresholds.

Observation I: PCC is dependent on the chosen criticality-rank threshold. During application execution, the instantaneous PCC(Th_{CR}) is a function of the chosen Th_{CR} value. For example, in `SCP`, values of PCC(Th_{CR} = 1) and PCC(Th_{CR} = 7) at a given time are very different. This follows from our previous discussion: as Th_{CR} increases, the number of critical cores increases, leading to high PCC values. The other three applications (`LUH`, `RAY`, `CONS`) also exhibit this trend.

Observation II: PCC varies within an application over time. Even for a fixed value of Th_{CR}, PCC(Th_{CR}) may not be constant throughout execution. For example, as observed prominently in `LUH` and `RAY`, PCC(Th_{CR} = 4) can be different over time, implying that the number of critical cores for the same value of Th_{CR} is not constant over time. In `CONS`, the change in PCC(Th_{CR}) over time is not very prominent, and PCC(Th_{CR} = 4) is in the mid-range (40-60%), implying that roughly half the cores are critical.

Observation III: PCC varies across applications. Across applications, even for the same value of Th_{CR}, PCC(Th_{CR}) can be very different. For example, at Th_{CR} = 4, `RAY` and `SCP` have very different PCC(Th_{CR} = 4) values (`SCP`'s is fairly higher than `RAY`'s). This indicates that, with Th_{CR} = 4, the number of critical cores for `SCP` is higher than that of `RAY`.

We also analyze the effect of memory bandwidth on the PCC metric. Figure 4 plots PCC[6] for varying amounts of main memory bandwidth (as described in Figure 1). We make one major observation:

Observation IV: PCC reduces significantly as main memory bandwidth increases. This trend is consistent with our discussions from Section 2 that the variation average memory latency observed by different cores (i.e., variation in criticality across different cores) reduces with higher memory bandwidth.

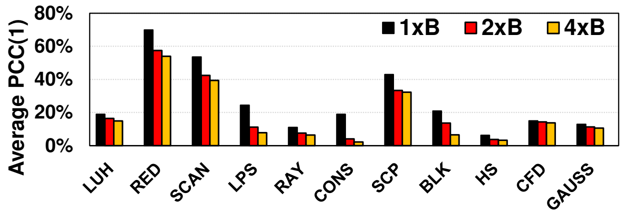


Figure 4: Effect of main memory bandwidth on PCC.

4. ANALYZING CRITICALITY IN THE MEMORY SYSTEM

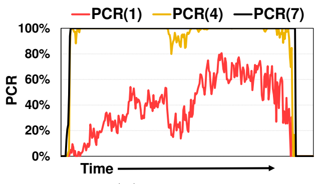
Cores with low latency tolerance are less likely to tolerate the latency of an outstanding memory request, making their requests more critical. Thus, our goal is to design a *criticality-aware* memory-scheduler that takes advantage of differences in criticality among requests and prioritizes the latency-critical requests to improve system performance. One of the important steps in designing such a memory scheduler is to gauge the variation in criticality across GPU cores. As discussed in Section 3.2, the PCC(Th_{CR}) metric is one of the key indicators of existence of different levels of criticality among cores, and in turn their corresponding memory requests. If the PCC(Th_{CR}) metric indicates that the latency tolerance variance across cores exists for a particular value of Th_{CR}, the memory scheduler can potentially prioritize requests from cores that have lower ranks. This is because such cores are more likely to have a large number of stalled warps that are waiting for memory requests to be serviced. We can prioritize requests from such cores to *proactively* avoid causing these cores to stall. However, note that as PCC(Th_{CR}) is dependent on Th_{CR}, we need to carefully examine the PCC(Th_{CR}) values for all possible

[6]We choose to show results for PCC with Th_{CR} = 1 because the critical cores at this Th_{CR} have the least latency tolerance, and reducing PCC(1) by prioritizing the memory requests of such cores is advantageous for improving overall performance (Section 3.1).

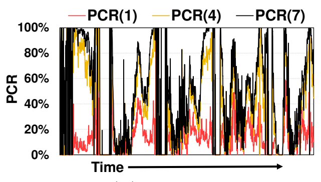
values of Th_{CR}, to understand at what level of Th_{CR} does substantial latency criticality variation across cores exist (if at all). If the PCC(Th_{CR}) metric does not indicate significant variance at any Th_{CR} level, the memory requests from different cores will have similar latency criticality. If this is the case, the memory scheduler can focus on preserving locality (like an existing GPU memory scheduler) since giving some requests higher priority than others is less likely to improve system performance.

The calculation of PCC(Th_{CR}) requires global information exchange across cores, and the hardware overhead of calculating this information and then communicating it directly to the memory controllers (MCs) periodically can be expensive. Instead, we propose to capture the variations in latency tolerance across cores directly at the MCs. To do so, we relay the current latency tolerance level of a core to the GPU memory scheduler by tagging the memory requests originating from that core with the core's current rank, and then calculating, at the MC, a metric called percentage of critical requests (PCR), which is defined as percentage of critical memory requests present in the MC memory request buffer. Again, because the decision of defining requests (or cores as we discussed before in Section 3.2) as critical is dependent on the value of Th_{CR}, we define PCR(Th_{CR}), which is the percentage of critical requests (*the requests that are tagged with rank values less than or equal to* Th_{CR}) in the MC request buffer. *Note that the observations discussed for PCC(Th_{CR}) in Section 3.2 hold true for PCR(Th_{CR}) as well. This is because the only difference of PCR(Th_{CR}) from PCC(Th_{CR}) is that PCR(Th_{CR}) considers criticality of requests instead of their corresponding cores.*

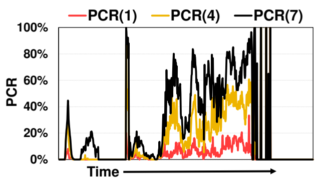
In Figure 5, we plot PCR(Th_{CR}) over time for the same four applications shown in Figure 3, for the same values of $Th_{CR} = \{1, 4, 7\}$. We observe that in both Figure 3 and Figure 5, applications have very similar patterns: PCR(Th_{CR} plots over time, shown in Figure 5, are highly correlated with the PCC(Th_{CR}) curves shown in Figure 3) Thus, a criticality-aware memory scheduler could make scheduling decisions based on PCR(k) $\forall k \in \{1...8\}$ information calculated locally at the MC, instead of using the global PCC(k), $\forall k \in \{1...8\}$ information.



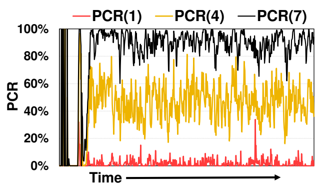
(a) SCP



(b) LUH



(c) RAY



(d) CONS

Figure 5: PCR metric over time with different criticality-rank thresholds.

Relaying each core's rank information to the MC, and then periodically calculating PCR(Th_{CR}) at the MC has two primary benefits: (1) this percentage can be calculated locally at the MC without requiring communication across MCs, as we just discussed, (2) the calculations to find the appropriate Th_{CR} value (discussed in Section 5) and other optimizations can also be done locally at the MCs.

Scope of Criticality Aware Scheduling. To understand the scope (i.e., potential opportunity) of criticality-aware memory scheduling in GPU workloads, we investigate the existence of memory requests with different criticality-ranks at the MCs at the same time. Figure 6 depicts the distribution of criticality-rank differences across DRAM requests, where criticality-rank difference is defined as the difference between the highest and the lowest criticality-rank of the memory requests present in the MC at the same time. This data is for one of the MCs (we observe similar distributions in other MCs), when more than one request is present in the MC buffer. In Figure 6, `diff-0` denotes the percentage of DRAM cycles during which *all* the memory requests in the buffer have the same criticality-rank. Similarly, `diff-1` denotes the percentage of DRAM cycles during which the difference between the highest and the lowest rank of the memory requests in the MC at the same instant is 1. Note that as the maximum possible rank is 8, the difference of the highest and the lowest rank can range from 0 to 7.

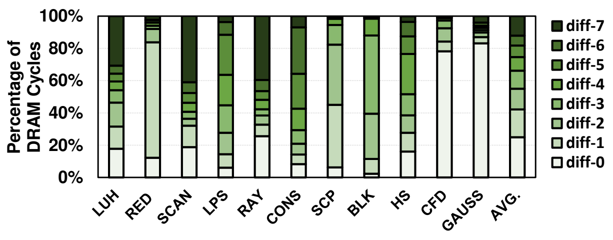


Figure 6: Distribution of criticality-rank differences across requests in the memory controller.

As we observe from Figure 6, the rank range present in the MC significantly varies across applications. In applications such as `LUH` and `RAY`, the difference in ranks is significant during most of the execution, while in other applications such as `GAUSS` and `CFD`, the difference is 0 most of the time. Therefore, in these applications, the scope of criticality-aware memory scheduling is likely to be lower. We observe that many applications (e.g., `RAY`, `LUH`, `CONS`, `RED`) have enough scope for criticality-based prioritization.

We conclude from these results that a memory scheduler that exploits the criticality differences across different cores' memory requests has promising scope to improve overall system performance.

5. DESIGN AND IMPLEMENTATION

5.1 Design Challenges of CLAMS

We identify two major challenges in designing CLAMS:

(I) Co-existence of critical and non-critical requests. In order to allow criticality-based prioritization, one of the important challenges is to find Th_{CR} such that both critical and non-critical requests coexist in the MC buffer. From our prior discussions, we observe that a high value of Th_{CR} might lead to *too* many cores and their corresponding requests to be considered as critical. On the other hand, a low value of Th_{CR} might lead to a *very* small number of cores and their corresponding requests to be considered as critical. In both scenarios, the MC buffer contains either only critical requests or only non-critical requests. This prevents the scheduler to take advantage of the differences between the ranks of the requests in the MC buffer. Therefore, *to increase the opportunities for criticality-based prioritization by distinguishing critical requests from others, it is imperative to find an appropriate Th_{CR} value that can enable substantial coexistence of both critical and non-critical requests in the system.*

(II) Balancing DRAM access locality and criticality. Even though choosing an appropriate Th_{CR} enables the co-existence of both critical and non-critical requests in the system, it might not achieve a good balance between exploiting locality (using the baseline scheduling policy that aims to maximize row buffer hit rate) and exploiting criticality (using our new policy that prioritizes critical requests). To address this trade-off, we periodically calculate PCR(Th_{CR}) to switch between scheduling policies optimized for criticality or locality. Over execution, if PCR(Th_{CR}) is *below* a threshold, which we call the *Scheduling-Mode-Threshold* (Th_{SM}), the scheduler prioritizes critical requests to ensure that the cores that cannot tolerate latency are serviced faster. However, if PCR(Th_{CR}) exceeds Th_{SM}, this implies that the differences in latency criticality have become insignificant and there are too many critical requests to prioritize one over another. In this case, the scheduler returns to the baseline mode of prioritizing row buffer hit requests. It is challenging to find an appropriate value for Th_{SM}, because a too high value for Th_{SM} would push the scheduler to serve critical requests for a longer time, potentially hampering locality. A too low value for Th_{SM} would not provide enough opportunities for criticality based prioritization. Therefore, it is important to find an appropriate Th_{SM} value to achieve a balance between exploiting locality and exploiting criticality.

Table 1 summarizes all metrics and thresholds we consider to design the CLAMS scheduler. We will make use of these in our descriptions in the rest of Section 5.

Table 1: List of CLAMS related metrics and thresholds.

Acronym	Description
PCC	Percentage of GPU cores in the critical state
PCR	Percentage of critical memory requests present in the MC request buffer
PCR_b	Ratio between the number of critical requests and the total number of requests, destined to b^{th} bank.
Th_{CR}	Specifies the rank up to which the core should be considered critical.
Th_{SM}	Specifies the threshold below which the scheduler prioritizes critical requests.

5.2 Design Overview of CLAMS

We propose three different schemes for calculating Th_{CR} and Th_{SM}. The first scheme is called as *Static-CLAMS* because it uses a *single and fixed* set of values for Th_{CR} and Th_{SM} for all applications. However, we find that these fixed and independent choices of Th_{CR} and Th_{SM} make it difficult to simultaneously address the two design challenges (Section 5.1). Therefore, our second scheme, called *Semi-Dyn-CLAMS*, dynamically calculates Th_{CR} based on: 1) a fixed value of Th_{SM}, and 2) PCR(k), $\forall k \in \{1...8\}$ information, calculated within an MC (Section 4). This scheme dynamically finds Th_{CR} but still uses a static value for Th_{SM}. Our third scheme is called *Dyn-CLAMS* because it dynamically calculates both Th_{CR} and Th_{SM}. It uses the same method as Semi-Dyn-CLAMS to calculate Th_{CR} and then dynamically updates Th_{SM} based on the calculated Th_{CR}. The thresholds calculated by these schemes are used to determine the *working mode* of CLAMS. The two working modes in which CLAMS scheduler can issue requests to the banks are *locality mode* and *criticality mode*.

Locality mode. This is the default scheduling mode in which CLAMS is locality-focused. It prioritizes: 1) row-hit requests over all other requests, 2) critical requests over other requests, 3) older requests over younger ones. Hence, if there are no critical requests present, this mode follows the baseline FR-FCFS scheduling policy [74, 93], which prioritizes: 1) row-hit requests over all other requests, 2) older requests over younger ones.

Criticality mode. In this mode, CLAMS is criticality-focused, and optimizes mainly for criticality. It prioritizes: 1) critical requests over all other requests, 2) row hit requests over other requests, 3) older requests over younger ones. Hence, if there are no critical requests present, this mode falls back to the baseline FR-FCFS policy.

Mode Selection. The decision to be in criticality or locality mode is based on the value of PCR calculated on a per bank basis, using the per-bank value $PCR_b(Th_{CR})$, which is defined as the ratio between the number of critical requests destined to b^{th} bank and the total number of requests destined to b^{th} bank. The particular mode is decided based on Eq.1.

$$PCR_b(Th_{CR}) \begin{cases} \leq Th_{SM} & \textit{criticality mode} \\ > Th_{SM} & \textit{locality mode} \\ = 0 & \textit{criticality} = \textit{locality mode.} \end{cases} \tag{1}$$

In the special case, when there are no critical requests destined to b^{th} bank ($PCR_b(Th_{CR}) = 0$), criticality and locality mode follow exactly the same request service order.

Inter-Core vs. Intra-Core Criticality. The goal of switching to the criticality mode is to prioritize critical memory requests over other requests belonging to *different* cores. However, because of the procedure we follow to tag criticality rank with a memory request, it might happen that both critical and non-critical requests from the **same** core might co-exist in an MC. As our schemes (explained next) do not explicitly consider core-ids while serving requests, it might happen that our prioritization mechanism might prefer a critical request over another; both belonging to the *same* core. This procedure has limited benefit, because requests from the same core have similar utility unless they have different *intra-core* criticality typically caused due to memory divergence [5, 11]. In our work, we are more interested in *inter-core* criticality, which is due to the fact that different cores have different levels of latency tolerance. Our detailed analysis shows that our schemes benefit more from *inter-core* criticality. On average, in criticality mode, 76% of the decisions to prioritize critical requests over non-critical ones are made for requests from different cores.

5.3 Design of the Static-CLAMS Memory Scheduler

This is the simplest of our proposed schemes. It uses fixed values for both the thresholds to identify the working mode. However, such fixed and independent choices for threshold values make the two design challenges of CLAMS harder to solve effectively (Section 5.1). To understand this, consider Figure 7, where we show the distribution of memory requests for one of the MCs (distribution for other MCs are similar) across different criticality-ranks when executed on our baseline architecture that employs FR-FCFS.

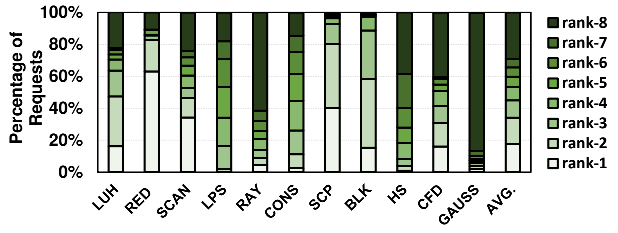


Figure 7: Distribution of requests in different criticality-rank states.

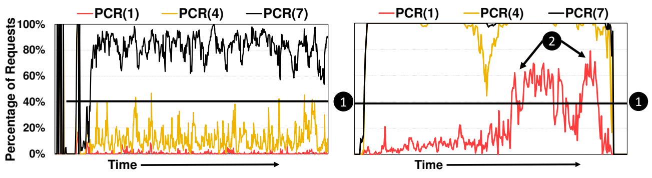
We observe, from the AVG. bar, that Th_{CR}=4 leads to co-existence of both critical and non-critical requests in the MC buffer. However, this value of Th_{CR} does not provide substantial coexistence in *every* application. For example in, SCP, with Th_{CR}=4, a majority of the requests are critical. Therefore, with this value of Th_{CR} along with Th_{SM}=20%, the scheduler will be in locality mode *most of the time*[7], as it will detect that there are too many critical requests in the MC. On the other hand, with Th_{SM}=80%, the scheduler will be mostly in criticality mode. This can degrade DRAM row buffer locality, leading to significant loss in performance. From this discussion, we conclude that: *(1) there is a need for adapting the Th_{CR} to the executing application, and (2) Th_{CR} and Th_{SM} should not be determined independently of each other.*

5.4 Design of the Semi-Dyn-CLAMS Memory Scheduler

The primary goal of Semi-Dyn-CLAMS is to calculate Th_{CR} with the help of a fixed Th_{SM} value and PCR(k) information, $\forall k \in \{1...8\}$[8]. This procedure achieves two additional sub-goals. First, Semi-Dyn-CLAMS makes Th_{CR} dependent on Th_{SM}, as it calculates Th_{CR} dynamically based on the fixed Th_{SM} value. Therefore, Semi-Dyn-CLAMS does *not* determine Th_{CR} and Th_{SM} independently, which is desirable based on the discussion in Section 5.3. Second, it makes Th_{CR} and Th_{SM} cognizant of *all* the requests in the MC request buffer, i.e., the PCR(k), $\forall k$ information. This is important because as PCR(k) and PCC(k) values are correlated (Section 4), making Th_{CR} and Th_{SM} aware of PCR(k), in turn, makes them aware of the current state of variation in criticality across cores.

To increase the opportunities of driving the memory scheduler in criticality mode, we dynamically find Th_{CR} such that the percentage of requests that are critical, denoted by PCR(Th_{CR}), is less than or equal to a fixed Th_{SM} value, but also as close as possible to Th_{SM}. In other words, we need to find the highest Th_{CR} such that PCR(Th_{CR}) is less than or equal to a fixed Th_{SM} value. After obtaining Th_{CR}, the scheduler will mostly be in criticality mode because we have ensured that over a window, PCR(Th_{CR}) is less than or equal to Th_{SM}. However, after being in criticality mode, if PCR (even with $Th_{CR} = 1$) exceeds the fixed value of Th_{SM}, the scheduler switches to locality mode, because our scheme detects that there are *too* many critical requests to prioritize and thus the latency tolerance variation across cores is not significant. Therefore, in such scenarios, we set Th_{CR}=8, which makes all requests to be considered critical. Such a value of Th_{CR} value will always drive the scheduler to locality mode, because PCR(8) is always equal to 1 and greater than Th_{SM}. We empirically find that Th_{SM} prefers to be in a mid-range (40% provided the best average performance results, Section 7). This is expected because, mid-range percentage of critical requests allows the coexistence of both critical and non-critical requests in MC.

Figure 8 illustrates the operation of Semi-Dyn-CLAMS over time, where we choose Th_{SM}=40% (❶). Assume only three values of Th_{CR} = {1,4,7} are possible for illustration purposes (in our final evaluation, we consider the full range from 1 to 8). We observe from Figure 8a that, in CONS, Th_{CR} is chosen as 4 most of the time because this value of Th_{CR} is the highest possible value of Th_{CR} such that PCR(Th_{CR}) is less than or equal to Th_{SM}=40%. Therefore, in CONS, the scheduler will mostly be in criticality mode. In SCP (Figure 8b), the situation is different. By choosing the same value of Th_{SM}=40% in the first half of the execution, the scheduler will be in criticality mode most of the time, as PCR(1) is lower than Th_{SM}=40%. However, during the second half of the execution, SCP prefers locality mode (❷), where PCR(1) line is above the horizontal line of Th_{SM}=40% (❶). During this time, there is no Th_{CR} such that PCR(Th_{CR}) is less than Th_{SM}=40%, and hence our scheme detects that there are too many critical requests (even with Th_{CR}=1). We set the scheduler to go in locality mode by setting Th_{CR}=8.



(a) CONS (b) SCP

Figure 8: Execution of CONS and SCP to illustrate the working of Semi-Dyn-CLAMS. Th_{CR} values are calculated dynamically.

Discussion. Recall that the actual mode selection is based on Eq.1, which compares the value of $PCR_b(Th_{CR})$ (and not PCR(Th_{CR})) with the value of Th_{SM}. Therefore, even though Semi-Dyn-CLAMS selects a value of Th_{CR} such that PCR(Th_{CR}) is lower than (or equal to) Th_{SM}, it is not necessary that $PCR_b(Th_{CR})$ will also be lower than (or equal to) to Th_{SM}. Therefore, even though Semi-Dyn-CLAMS overall strives to keep the scheduler in the criticality mode, while ensuring that both critical and non-critical requests have substantial presence in MC, during actual issue of requests to the memory banks, the scheduler can be in any of the modes – locality or criticality. However, if, $\forall k \in \{1..8\}$, PCR(k) is greater than the Th_{SM} value, the scheduler switches to the locality mode by setting the Th_{CR}

[7]Note that the actual working mode selection is based on Eq. 1, where $PCR_b(Th_{CR})$ (and not PCR(Th_{CR})) is used. Thus, the actual decision of being in criticality mode or locality mode is made on a per-bank basis. Section 5.4 provides more details.

[8]This information is updated every 512 cycles. We also used three other sampling size windows (256, 1024, 2048) cycles. The difference in overall average performance is less than 1%, implying that sampling window size does not have a significant impact on our design.

value to 8. This value of 8 will always switch the scheduler to the locality mode, because, by definition, both PCR(8) and $PCR_b(8)$ are always equal to 1, and therefore, it will be always greater than Th_{SM}. We use this analysis as a foundation for our next scheme.

Importance of $PCR_b(Th_{CR})$ and PCR(k)$\forall k$. For calculating the appropriate Th_{CR}, Semi-Dyn-CLAMS consults PCR(k), $\forall k$ information, but the scheduler makes actual decisions on the working modes based on the current set of requests to be issued to the bank, i.e., by examining $PCR_b(Th_{CR})$. This has two advantages. First, the actual mode decision is aware of the current state of the requests destined to each bank. Second, this decision is also aware of the status of all the requests in the MC, which, in turn, is also aware of PCC information.

5.5 Design of Dyn-CLAMS Memory Scheduler

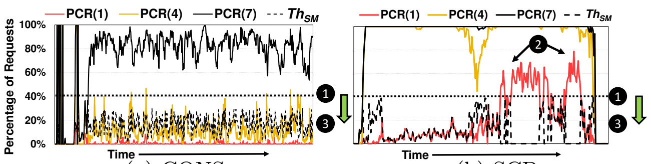
We find Semi-Dyn-CLAMS to be an aggressive design in taking advantage of criticality because of two reasons. First, Semi-Dyn-CLAMS always strives to find opportunities to work in criticality mode. Second, Semi-Dyn-CLAMS goes into locality mode only when there are too many critical requests at Th_{CR}=1 (PCR(1) $> Th_{SM}$) in the MC buffer. Due to these two reasons, we observe significant loss in locality and performance in some applications (e.g., `SCP` and `RAY`) where locality is very important.

Even though Semi-Dyn-CLAMS calculates a Th_{CR} value that facilitates the scheduler to be in criticality mode, when it actually issues requests to the bank, the scheduler might actually pick locality mode based on the $PCR_b(Th_{CR})$ value for each bank (Section 5.4). The goal of Dyn-CLAMS is to improve locality by increasing such opportunities.

To do so, we eliminate one of the important limitations of Semi-Dyn-CLAMS: that it still uses a fixed value of Th_{SM}. In other words, in Semi-Dyn-CLAMS, Th_{CR}-Th_{SM} dependence is only *one way*, and Th_{SM} is *not* updated based on the calculated Th_{CR} value. Therefore, the *key idea* of Dyn-CLAMS is to first gauge the negative effect of the loss in row-locality on the latency tolerance (i.e., performance) of the cores by dynamically examining Th_{CR}, and then restoring the loss by lowering the value of Th_{SM} as much as possible while maintaining the same value of Th_{CR} calculated using Semi-Dyn-CLAMS. This is because, with a lower value of Th_{SM}, the scheduler is more likely to work in locality mode (see Eq. 1).

Dyn-CLAMS uses exactly the same procedure as adopted by Semi-Dyn-CLAMS to calculate Th_{CR}, but in addition, also lowers Th_{SM}. At the beginning of every window, we start with a fixed Th_{SM} value (Th_{SMinit}) to determine Th_{CR} using Semi-Dyn-CLAMS. The value of Th_{SMinit} is equal to 40%, which we calculated based on extensive experimental evaluation. After calculating Th_{CR}, we update (lower) the value of Th_{SM} to PCR(Th_{CR}). By doing so, Th_{CR} remains the same as per Semi-Dyn-CLAMS scheme, but Th_{SM} is reduced. Thus, both Th_{CR} and Th_{SM} values are updated dynamically.

Figure 9 illustrates this scheme. For `CONS`, we observed in Semi-Dyn-CLAMS (Figure 8) that Th_{CR} is usually 4, but in Dyn-CLAMS, while maintaining the same Th_{CR}, the value of Th_{SM} is lowered (pointed by ❶ $\rightarrow$ ❸) such that it closely resembles the PCR(4) curve. Similarly, in `SCP`, the value of Th_{SM} is lowered to match closely with PCR(1). However, in cases when SCP prefers locality mode (❷), Semi-Dyn-CLAMS sets Th_{CR} to 8, and Dyn-CLAMS sets Th_{SM} to 0, making the scheduler work in locality mode. Table 2 formally describes the procedures adopted by our schemes.



(a) CONS (b) SCP

Figure 9: Execution of CONS and SCP to illustrate the working of Dyn-CLAMS. Th_{SM} is dynamically updated based on Th_{CR}.

5.6 Hardware Overheads

We describe the hardware overheads of the three flavors of CLAMS. All three flavors requires logic to tag a memory request with the criticality of the core that generates it.

Tagging memory requests with core criticality. Each core is assigned with a rank depending on its current state of latency tolerance. As the maximum possible number of warps on a core is 48, we need two 13-bit counters to store the windowed-average of short-latency and issued warps over 128 cycles. We calculate the rank using one 13-bit divider and eight comparators. This rank is stored in a 3-bit register. At the time when a memory request is issued, we tag the memory request with the corresponding core's rank.

(I) Static-CLAMS. Two 8-bit up-down counters per-bank (max. MC buffer size is 256) are required to track the number of critical and pending memory requests. For mode selection, we compare the value of $PCR_b(Th_{CR})$ to a fixed value of Th_{SM} with the help of comparator. Th_{CR} (3 bits) and Th_{SM} (7 bits) values are stored as fixed thresholds in registers at the MC.

(II) Semi-Dyn-CLAMS. We calculate PCR(k) $\forall k$ per MC over a window of 512 cycles by keeping track of the critical requests and the number of total requests at the MC. We need one 9-bit counter per rank and one 9-bit counter

Table 2: Pseudo code for our proposed schemes

```
Semi-Dyn-CLAMS (Section 5.4)
    $Th_{CR} = 8$.
    for $k \in \{1...7\}$ do
        if ($0 < PCR(k) \leq Th_{SM} < PCR(k+1)$) then
            $Th_{CR} = k$.
            end if; end for
    return $Th_{CR}$.
```

```
Dyn-CLAMS (Section 5.5)
    $Th_{SM} = Th_{SMinit}$; $Th_{CR} = 8$.
    for $k \in \{1...7\}$ do
        if ($0 < PCR(k) \leq Th_{SM} < PCR(k+1)$) then
            $Th_{CR} = k$; $Th_{SM} = PCR(Th_{CR})$.
            end if; end for
    if $Th_{CR} = 8$ then $Th_{SM} = 0\%$. end if
    return ($Th_{CR}$, $Th_{SM}$).
```

Table 3: Key configuration parameters of the simulated GPU configuration.

Core Features	1400MHz core clock, 32 cores (streaming multi-processors), SIMT width = 32 (16 × 2), Greedy-then-oldest first (GTO) dual warp scheduler [67], Thread-blocks are scheduled on SMs in a load-balanced fashion
Resources / Core	48KB shared memory, 32KB register file, Max. 1536 threads (48 warps, 32 threads/warp)
Private Caches / Core	16KB 4-way L1 data cache, 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
Shared L2 Cache	16-way 128 KB/memory channel (768KB in total), 128B cache block size
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Memory Model	6 GDDR5 Memory Controllers (MC), FR-FCFS scheduling, 256 max. common request buffer for all 8 banks per MC, 924 MHz memory clock Global address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing, $t_{CL}=12$, $t_{RP}=12$, $t_{RC}=40$, $t_{RAS}=28$, $t_{CCD}=2$, $t_{RCD}=12$, $t_{RRD}=6$, $t_{CDLR}=5$, $t_{WR}=12$
Interconnect	1 crossbar/direction (32 cores, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

to keep track of the pending memory requests. We take a snapshot of these counters in extra storage, and then flush the counters. We then calculate PCR(k) $\forall k$ based on the snapshot values, and store them in 8 PCR(k) registers. To calculate Th_{CR}, we compare the fixed Th_{SM} value with 8 PCR(k) registers.

(III) Dyn-CLAMS. This scheme updates Th_{SM} with the value of PCR(Th_{CR}), Thus, it does not require extra overhead over Semi-Dyn-CLAMS. The information for all schemes is computed locally at the MCs. The total storage required for one of the MCs is 43B.

6. EVALUATION METHODOLOGY

We simulate the baseline architecture described in Table 3 using GPGPU-Sim v3.2.1 [6], a cycle-accurate GPU simulator. We studied 11 applications (Table 4) from various suites such as SDK [66], Rodinia [12], LLNL [42], and SHOC [13]. We classify these applications into two classes. Class-A applications show high-to-moderate scope for criticality-aware scheduling because of the presence of variation in criticality across cores (see Figure 1). The other applications are classified as Class-B because of low scope for criticality-aware scheduling (see Figure 6 and Figure 1). All the applications are executed until completion, except for `LUH`, `CONS`, and `CFD`, where we execute 500 million instructions.

Table 4: Evaluated applications. Table also shows: 1) Average occupancy (occ) of a GPU core in terms of warps, 2) Average Th_{CR} and Th_{SM} calculated using Semi-Dyn-CLAMS and Dyn-CLAMS, respectively, and 3) % of critical requests served in criticality-mode with Dyn-CLAMS (%-cri).

Application	Abbr.	Class	occ	Th_{CR}	Th_{SM}	%-cri
Lulesh [42]	LUH	A	18	4	13%	46
Reduction [13]	RED	A	15	1	16%	38
Scan [13]	SCAN	A	14	4	16%	42
Laplace 3D [66]	LPS	A	11	4	23%	32
Ray Tracing [66]	RAY	A	16	6	11%	40
Convolution [66]	CONS	A	15	5	21%	45
Scalar Product [66]	SCP	B	30	1	10%	66
BlackScholes [66]	BLK	B	32	2	8%	70
Hotspot [12]	HS	B	23	6	17%	17
CFD Solver [12]	CFD	B	45	2	4%	10
Gaussian [12]	GAUSS	B	8	6	6%	2

7. EXPERIMENTAL RESULTS

In this section, we analyze the performance of three CLAMS designs, along with two more memory schedulers: FR-FCFS-Cap-Best and Static-CLAMS-Best. FR-FCFS-Cap (streak control) scheduler [62] enforces a *cap* on the younger row-hit requests that can be serviced before an older row request to the same bank. When the cap is reached, FCFS policy is applied. While such a cap alleviates the starvation problem for waiting requests, it is not aware of the criticality of requests it is servicing. We show the results of FR-FCFS-Cap-Best that picks the best performing *cap* threshold, profiled separately for *each* application. Evaluated choices for *cap* values are 2, 4, 6, 8, 12 and 16. Static-Best-CLAMS is the Static-CLAMS scheduler that uses the best performing combination of Th_{CR} and Th_{SM}, profiled separately for *each* application. In contrast to Static-Best-CLAMS, Static-CLAMS uses a *single* set of thresholds (Th_{CR}=4 and Th_{SM}=20%) that provides the best *average* performance across *all* applications. The values of these thresholds are chosen from a pool of 42 (7 × 6) different combinations formed using fixed values of Th_{CR} (1 through 7) and Th_{SM} (0% through 100% in steps of 20%). Note that both FR-FCFS-Cap-Best and Static-Best-CLAMS are hard to implement as they require an exhaustive search across many threshold combinations on a per application basis. We observe that the FR-FCFS-Cap results are very sensitive to thresholds and a single threshold does not work well for all the applications. Dynamic adaptation of such thresholds is non-trivial and that is why we propose CLAMS.

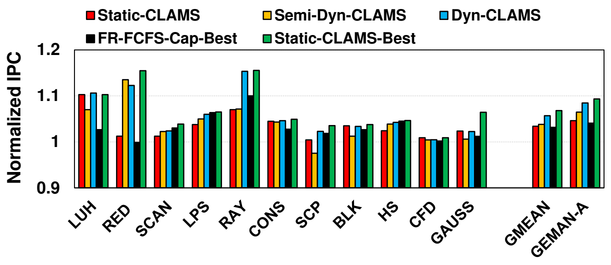
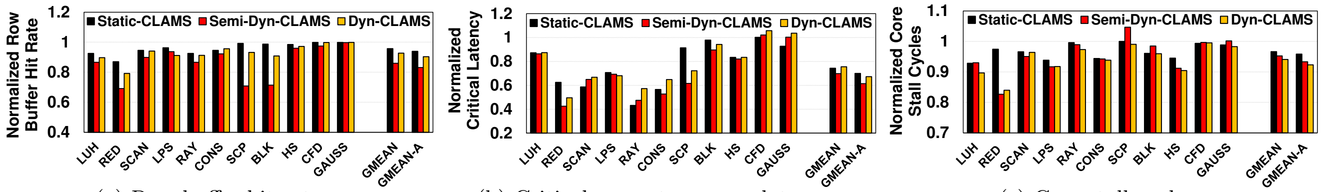


Figure 10: Performance results normalized to FR-FCFS.

Figure 10 shows the IPC improvement over FR-FCFS of proposed memory schedulers. We provide two averages, GMEAN for all applications, and GMEAN-A for only Class-A applications. We also present auxiliary metrics related to DRAM and GPU cores in Figure 11. Figure 11a shows the Row Buffer Hit Rate (RBHR) to measure DRAM locality. Figure 11b depicts the latency of critical memory requests (with respective values of Th_{CR}), and Figure 11c shows the core stall cycles during which GPU cores are not able to issue any warps. Reductions in critical request latencies and



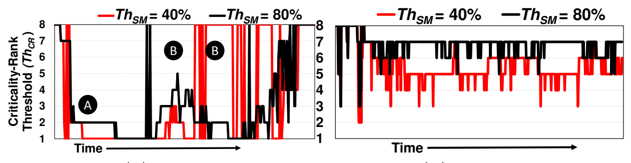
(a) Row buffer hit rates (b) Critical request memory latency (c) Core stall cycles

Figure 11: (a) DRAM row buffer hit rate, (b) memory latency of critical requests, (c) core stall cycles. Results are normalized to FR-FCFS.

core stall cycles are attributed to our prioritization schemes which favor critical requests. All results are normalized to the baseline FR-FCFS scheduler.

Analysis of Static-CLAMS. Using a single set of thresholds (Th_{CR}=4 and Th_{SM}=20%) for all applications does not lead to significant improvements over FR-FCFS in `SCP` and `RED`. This is expected because with Th_{CR}=4, a high percentage of requests are treated as critical, and using Th_{SM}=20% along with it pushes the scheduler to work mostly in locality mode. However, for `LUH`, we observe 10% IPC improvement, because static thresholds address both the design challenges reasonably (Section 5.1). On average, Static-CLAMS provides 3% IPC improvement for all 11 applications. Although none of the applications experience performance degradation, this scheme is still far from the upper-bound performance achievable with Static-Best-CLAMS.

Analysis of Semi-Dyn-CLAMS. We first analyze the dynamic changes in Th_{CR} calculated by the Semi-Dyn-CLAMS scheme for two applications – `SCP` and `CONS`. Figure 12 shows these results for two fixed values (40%, 80%) of Th_{SM}. We first start when Th_{SM}=40%. In `SCP`, Semi-Dyn-CLAMS chooses Th_{CR}=1 in the first half of the execution (Ⓐ) (as expected from our discussion in Section 5). In the second half of the execution (Ⓑ), we observe many switches to Th_{CR}=8 as Semi-Dyn-CLAMS detects that there are too many critical requests and hence switches to locality mode. In `CONS`, our scheme chooses Th_{CR} between 4 and 5 and mostly remains in criticality mode. However, when Th_{SM} is set to 80%, we observe an increase in Th_{CR} values for SCP in the first half of the execution (Ⓐ), and Semi-Dyn-CLAMS switches to locality mode less often (because fewer instances are observed where Th_{CR}=8). This is expected because with a higher Th_{SM} value, the scheduler will switch to criticality mode more often.

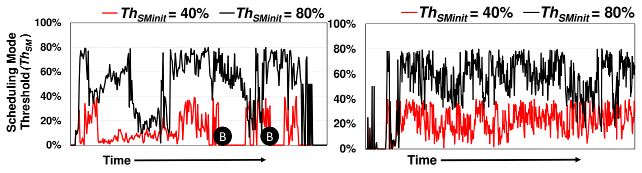


(a) SCP (b) CONS

Figure 12: Changes in Th_{CR} observed with Semi-Dyn-CLAMS. When Th_{CR}=8, the scheduler is in locality mode.

On average, Semi-Dyn-CLAMS provides 5% IPC improvement over FR-FCFS. `RED`, `RAY`, and `LPS` experience 13%, 7%, and 5% improvement, respectively. As desired, this scheme attempts to push the scheduler mostly to criticality mode, which helps to reduce the latency of critical requests further by 6% (at the cost of 10% reduction in RBHR) compared to Static-CLAMS. This in turn reduces the core stall cycles further by 2% compared to Static-CLAMS (Figure 11c). In `SCP`, row buffer hit rate is hampered the most, by 30% (Figure 11a), leading to 5% performance degradation compared to FR-FCFS. On the other hand, in `RED`, even though RBHR is reduced by 20%, performance improves significantly (by 12%) due to the reduction in critical request latency and core stall cycles. Our detailed analysis shows that, in `RED`, the impact of locality on performance is much lower than that of criticality, and vice versa in `SCP`. Our next scheme, Dyn-CLAMS is expected to recover the loss in locality and performance by reducing the Th_{SM} value dynamically.

Analysis of Dyn-CLAMS. We first analyze the dynamic changes in Th_{SM} calculated by the Dyn-CLAMS scheme for two applications – `SCP` and `CONS`. Figure 13 shows these changes when Th_{SMinit} = 40% and 80%. We start with analyzing the Th_{SM} curves when Th_{SMinit}=40%. In `SCP` and `CONS`, we find that the value of Th_{SM} is less than or equal to 40%. This helps the scheduler to switch to locality mode more frequently, as discussed in Section 5.5. During the phases when Th_{CR}=8 (as pointed in Figure 12 by Ⓑ), Th_{SM} value is 0% (Ⓑ), pushing the scheduler to always be in locality mode. For Th_{SMinit} = 80% curves, we observe that the value of Th_{SM} is much higher due to the increase in Th_{CR} values.



(a) SCP (b) CONS

Figure 13: Changes in Th_{SM} observed with Dyn-CLAMS.

On average, Dyn-CLAMS performs better than all three memory scheduling schemes except the upper bound Static-Best-CLAMS. `RED`, `RAY`, and `LUH` are the best performers with 15%, 15%, and 10% improvement over FR-FCFS, respectively. This scheme is especially useful for applications where exploiting locality is at least as important as exploiting criticality. For example, in `SCP` and `RAY`, RBHR is improved by 5% and 7%, respectively (Figure 11a), leading to additional benefits over Semi-Dyn-CLAMS. We also observe reduction in PCC(1) for these applications (22%, 25%, and 6%, respectively), as expected from our discussion in Section 3.2.

We further observe from Figure 10 that the gap between Dyn-CLAMS and Static-CLAMS-Best is not significant for many applications, suggesting that Dyn-CLAMS is able to dynamically calculate the best static combinations of thresholds for each application, as shown in Table 4, without the need for any offline application profiling. We also report the percentage of critical requests that are served in criticality mode (%-cri) in Table 4. For SCP and BLK, even though %-cri is very high, Dyn-CLAMS does not show benefit because the *number* of critical requests is small. For Class-A applications, %-cri is significant, which shows that Dyn-CLAMS is able to improve performance by prioritizing critical requests. In summary, Dyn-CLAMS achieves 9% IPC improvement over FR-FCFS, 5% over FR-FCFS-Cap-Best, and also is within 1% of the Static-CLAMS-Best for Class-A applications. The performance of none of the Class-B applications degrades with our third, most dynamic scheme.

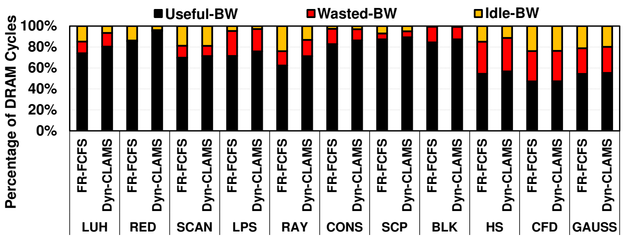


Figure 14: Useful, wasted, and idle DRAM bandwidth distribution with baseline FR-FCFS ad Dyn-CLAMS.

To get a deeper understanding into these performance results, Figure 14 shows the break down of the memory bandwidth for FR-FCFS and Dyn-CLAMS schemes into the following components: (A) *Useful-BW:* the percentage of DRAM cycles during which the application moves data (reads and writes) over the DRAM interface, (B) *Wasted-BW:* the percentage of DRAM cycles during which no data is transferred over the DRAM interface due to DRAM timing constraints [9], but there are pending memory requests in the MC buffer, and (C) *Idle-BW:* the percentage of DRAM cycles during which there are no requests pending in the MC buffer, and hence DRAM is idle. We observe that IPC and Useful-BW are highly correlated, which is consistent with the findings shown by Guz et al. [31, 32]. We further observe that Wasted-BW increases in LUH and RAY with Dyn-CLAMS because of the loss in RBHR. This is expected as the loss in locality causes more row conflicts. However, in SCAN, in spite of the reduction in RBHR, we observe negligible reduction in Wasted-BW. This is because Dyn-CLAMS enables more cores to be active at a time, allowing requests from more cores to take advantage of BLP. This increase in BLP helps to mask the negative effects of the loss in locality.

Brief Summary of Sensitivity Studies. For Semi-Dyn-CLAMS, increasing the Th_{SM} value from 20% to 40% improves the performance of applications (e.g., CONS, RED) that prefer the criticality mode. However, beyond Th_{SM}=40%, performance of most applications (except RED) saturates. Performance starts declining after Th_{SM}=60% due to the steep decrease in RBHR. We observe similar trends for Th_{SMinit} in Dyn-CLAMS. On average, a value of 40% for both Th_{SM} and Th_{SMinit} leads to the best average performance results across all our applications.

8. RELATED WORK

To our knowledge, this is the first paper that develops (1) the notion of *core criticality* in GPU systems, (2) mechanisms to dynamically estimate core criticality for different GPU cores, and (3) techniques that use such criticality estimation mechanisms to improve GPU performance via sophisticated memory request scheduling. As such, this paper is related to past works on criticality estimation in CPUs, memory scheduling (for both GPUs and CPUs), and in general other scheduling mechanisms in GPUs. We briefly summarize the prior work related to this paper in these broad categories.

Criticality Related Studies in CPUs. There have been several works that identified critical instructions [10, 21–23, 78, 79, 81], critical threads [8, 9, 17, 20, 36, 37, 86], critical applications [14, 18, 47, 48, 63], critical memory requests [4, 26, 33, 34, 45, 51, 70, 87], and critical network packets [15, 16, 29, 30], within the context of a CPU system, and developed mechanisms to analyze and/or prioritize them. None of these works identify or exploit the notion of *core criticality* we describe and analyze in this paper, which is specific to GPU systems. Srinivasan et al. [78] contrasts criticality and locality by performing a limit study that gives the maximum potential of exploiting critical memory requests. A follow-up work [79] gives a practical algorithm for identifying critical memory requests within the context of CPUs that employ out-of-order execution and explores caching mechanisms for exploiting both criticality and locality. Other works, mentioned above, take advantage of criticality information of various entities, including instructions, threads, applications, memory requests, and network packets, by developing different notions of criticality. In comparison, the criticality notion we introduce and exploit in this paper is based on the latency tolerance of a GPU core, which is very different from that of prior works.

Memory Scheduling in CPUs. Several works exploited various notions of criticality for memory request scheduling in CPU systems. Ebrahimi et al. [20] propose parallel application memory scheduling, where the memory scheduler prioritizes critical threads in multi-threaded CPU programs, where critical threads are estimated to be those that are likely on the critical path of execution. Ghose et al. [26] use load criticality information for effective memory scheduling in CPU systems. In contrast to their static memory controller policy that always favors critical requests, we develop memory scheduling mechanisms that can *dynamically* switch between criticality and locality modes. Ausavarungnirun et al. [4] propose a memory scheduler to maintain row-buffer locality in CPU-GPU fused architectures while providing high CPU performance. Our insights can be combined with the insights of this work: if we want to use our techniques for GPUs present in the CPU-GPU fused architectures [4, 44], we should also consider latency tolerance of CPU cores and include that information in the prioritization mechanisms at the memory controller.

There are numerous other memory scheduling techniques [19, 33, 46–48, 52, 53, 60–63, 65, 71, 76, 82–85, 87, 92] designed for various goals in multi-core and multi-threaded systems. None of these works are designed for GPUs and, hence, their notion of request criticality, when it exists, is very different from our notion of core criticality.

Memory Scheduling in GPUs. Lakshminarayana et al. [50] explored a DRAM scheduling policy that chooses between two policies: Shortest Job First (SJF) and FR-FCFS. Their scheme uses a statically determined parameter that needs to be uniquely calculated for each application. As this scheduler does not adapt to dynamic needs of criticality and locality in the application, it can potentially degrade per-

[9] Note that, improving RBHR and bank-level parallelism (BLP) can reduce this Wasted-BW.

formance compared to FR-FCFS [50]. On the other hand, CLAMS *dynamically* detects the preferred mode (locality or criticality) for applications at runtime and switches between two modes: Criticality and Locality. Due to its heavily dynamic nature in adapting thresholds used to workload characteristics, we find that our mechanism does not degrade any workload's performance. Yuan et al. [91] propose an arbitration mechanism in the interconnection network to restore the lost row buffer locality caused by the interleaving of requests. They showed that performance of in-order DRAM is competitive to FR-FCFS. In this paper, we show qualitatively and quantitatively that CLAMS outperforms FR-FCFS. Ausavarungnirun et al. [5] develop a technique that can prioritize some warps over others based on the latency tolerance characteristics of the warps. Our core-level criticality based mechanisms can be combined with such warp-level criticality based mechanisms for higher performance improvements.

Warp Scheduling in GPUs. Narasiman et al. [64] and Gebhart et al. [25] proposed two-level warp schedulers to improve latency tolerance and energy consumption in GPUs, respectively. Rogers et al. [75] and Jog et al. [41] proposed warp schedulers to reduce contention in caches. Lee et al. [54] proposed a criticality-aware warp scheduler that prefers critical warps over others for better latency tolerance. None of these works specifically coordinate with the underlying memory schedulers for orchestrated warp and memory scheduling decisions. CLAMS provides a substrate to foster such research, as it incorporates the core-criticality information while making memory scheduling decisions.

9. CONCLUSIONS

We introduce the notion of core criticality in GPUs, a measure of latency tolerance of a GPU core, and a new GPU memory scheduler, called CLAMS, which exploits this new measure to improve GPU performance. Our scheduler takes into account the latency tolerance of GPU cores and prioritizes memory requests from such cores, when doing so would be beneficial. CLAMS dynamically determines the importance of exploiting core criticality versus exploiting DRAM locality by monitoring the fraction of critical requests in the memory request buffers. As such, it can adapt well to changing workload demands by swiftly switching between two scheduling policies optimized for criticality and locality. Our evaluations show that CLAMS provides significant performance benefits for the class of applications that exhibit high variance in criticality across cores, without hurting the performance of other applications. We conclude that considering core criticality is a promising way to improve GPU performance and hope future works can take advantage of the notion of core criticality for other system optimizations. In particular, we believe a promising area of research is to explore this notion within the context of heterogeneous CPU-GPU memory systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This research is supported in part by NSF grants #1205618, #1213052, #1212962, #1302225, #1302557, #1317560, #1320478, #1320531, #1409095, #1409723, #1439021, #1439057, and #1526750. Adwait Jog also acknowledges the start-up grant from College of William and Mary.

REFERENCES

[1] M. Abdel-Majeed and M. Annavaram, "Warped Register File: A Power Efficient Register File for GPGPUs," in *HPCA*, 2013.

[2] M. Abdel-Majeed *et al.*, "Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs," in *MICRO*, 2013.

[3] M. Arora *et al.*, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, 2012.

[4] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[5] R. Ausavarungnirun *et al.*, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.

[6] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[7] M. Bauer *et al.*, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization," in *SC*, 2011.

[8] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *ISCA*, 2009.

[9] Q. Cai *et al.*, "Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions," in *PACT*, 2008.

[10] J. Casmira and D. Grunwald, "Dynamic Instruction Scheduling Slack," in *Kool Chips Workshop in conjunction with MICRO*, 2000.

[11] N. Chatterjee *et al.*, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC*, 2014.

[12] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[13] A. Danalis *et al.*, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *GPGPU*, 2010.

[14] R. Das *et al.*, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.

[15] R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.

[16] R. Das *et al.*, "Aérgia: exploiting packet latency slack in on-chip networks," in *ISCA*, 2010.

[17] K. Du Bois *et al.*, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM.

[18] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," in *ASPLOS*, 2010.

[19] E. Ebrahimi *et al.*, "Prefetch-Aware Shared Resource Management for Multi-Core Systems," in *ISCA*, 2011.

[20] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *MICRO*, 2011.

[21] B. Fields *et al.*, "Slack: maximizing performance under technological constraints," in *ISCA*, 2002.

[22] B. Fields *et al.*, "Focusing processor policies via critical-path prediction," in *ISCA*, 2001.

[23] B. A. Fields *et al.*, "Interaction cost and shotgun profiling," 2004.

[24] J. Gai *et al.*, "More IMPATIENT: A Gridding-accelerated Toeplitz-based Strategy for non-Cartesian High-resolution 3D MRI on GPUs," *JPDC*, 2013.

[25] M. Gebhart *et al.*, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.

[26] S. Ghose *et al.*, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.

[27] N. Goswami *et al.*, "Power-performance Co-optimization of Throughput Core Architecture using Resistive Memory," in *HPCA*, 2013.

[28] N. Goswami *et al.*, "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications," in *IISWC*, 2010.

[29] B. Grot *et al.*, "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees," in *ISCA*, 2011.

[30] B. Grot *et al.*, "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip," in *MICRO*, 2009.

[31] Z. Guz *et al.*, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE CAL*, 2009.

[32] Z. Guz *et al.*, "Threads vs. Caches: Modeling the Behavior of Parallel Workloads," in *ICCD*, 2010.

[33] E. Ipek *et al.*, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.

[34] M. K. Jeong *et al.*, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *DAC*, 2012.

[35] W. Jia *et al.*, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *ICS*, 2012.

[36] J. A. Joao *et al.*, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *ASPLOS*, 2012.

[37] J. A. Joao *et al.*, "Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *ISCA*, 2013.

[38] A. Jog *et al.*, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *GPGPU*, 2014.

[39] A. Jog *et al.*, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.

[40] A. Jog *et al.*, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[41] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[42] I. Karlin *et al.*, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *IPDPS*, 2013.

[43] O. Kayiran *et al.*, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[44] O. Kayiran *et al.*, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.

[45] S. Khan *et al.*, "Improving Cache Performance by Exploiting Read-Write Disparity," in *HPCA*, 2014.

[46] H. Kim *et al.*, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS*, 2014.

[47] Y. Kim *et al.*, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[48] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[49] D. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[50] N. B. Lakshminarayana *et al.*, "DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function," *IEEE CAL*, 2012.

[51] C. J. Lee *et al.*, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.

[52] C. J. Lee *et al.*, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.

[53] C. J. Lee *et al.*, "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," in *HPS Technical Report*. University of Texas at Austin, 2010.

[54] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *PACT*, 2014.

[55] S.-Y. Lee and C.-J. Wu, "Characterizing GPU Latency Hiding Ability," in *ISPASS*, 2014.

[56] V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *ISCA*, 2010.

[57] J. Leng *et al.*, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[58] D. Li *et al.*, "Priority-Based Cache Allocation in Throughput Processors," in *HPCA*, 2015.

[59] E. Lindholm *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.

[60] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.

[61] S. P. Muralidhara *et al.*, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.

[62] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[63] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[64] V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.

[65] K. J. Nesbit *et al.*, "Fair Queuing Memory Systems," in *MICRO*, 2006.

[66] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011. http://developer.nvidia.com/cuda-cc-sdk-code-samples

[67] NVIDIA, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," 2011.

[68] G. Pekhimenko *et al.*, "A Case for Toggle-Aware Compression for GPU Systems," in *HPCA*, 2016.

[69] B. Pichai *et al.*, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014.

[70] M. K. Qureshi *et al.*, "A Case for MLP-Aware Cache Replacement," ser. ISCA, 2006.

[71] N. Rafique *et al.*, "Effective Management of DRAM Bandwidth in Multicore Processors," in *PACT*, 2007.

[72] M. Rhu *et al.*, "A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in *MICRO*, 2013.

[73] S. Rixner, "Memory Controller Optimizations for Web Servers," in *MICRO*, 2004.

[74] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.

[75] T. G. Rogers *et al.*, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[76] V. Seshadri *et al.*, "The dirty-block index," in *ISCA*, 2014.

[77] A. Sethia *et al.*, "APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency," in *PACT*, 2013.

[78] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *MICRO*, 1998.

[79] S. Srinivasan *et al.*, "Locality vs. Criticality," in *ISCA*, 2001.

[80] S. S. Stone *et al.*, "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distrib. Comput.*, 2008.

[81] S. Subramaniam *et al.*, "Criticality-Based Optimizations for Efficient Load Processing," in *HPCA*, 2009.

[82] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.

[83] L. Subramanian *et al.*, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *IEEE TPDS*, 2016.

[84] L. Subramanian *et al.*, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.

[85] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

[86] M. A. Suleman *et al.*, "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," in *ASPLOS*, 2009.

[87] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," in *ACM TACO*, 2016.

[88] N. Vijaykumar *et al.*, "Enabling Efficient Data Compression in GPUs," in *ISCA*, 2015.

[89] J. Wang and Y. Sudhakar, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *IISWC*, 2014.

[90] X. L. Wu *et al.*, "Impatient MRI: Illinois Massively Parallel Acceleration Toolkit for image reconstruction with enhanced throughput in MRI," in *ISBI*, 2011.

[91] G. Yuan *et al.*, "Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures," in *MICRO*, 2009.

[92] J. Zhao *et al.*, "FIRM: Fair and high-performance memory control for persistent memory systems," in *MICRO*, 2014.

[93] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order, US Patent," 1997.