Exploiting Latency and Error Tolerance of GPGPU Applications for an Energy-efficient DRAM

Haonan Wang and Adwait Jog
College of William & Mary
Email: hwang07@email.wm.edu, ajog@wm.edu

Abstract

Memory (DRAM) energy consumption is one of the major scalability bottlenecks for almost all computing systems, including throughput machines such as Graphics Processing Units (GPUs). A large fraction of DRAM dynamic energy is spent on fetching the data bits from a DRAM page (row) to a small-sized hardware structure called as the row buffer. The data access from this row buffer is much less expensive in terms of energy and latency. Hence, it is preferred to reuse the buffered data as much as possible before activating another row and bringing its data to these row buffers. Our thorough characterization of several GPGPU applications shows that these row buffers are poorly utilized leading to sub-optimal energy consumption. To address this, we propose a novel memory scheduling for GPUs that exploits latency and error tolerance properties of GPGPU applications to reduce row energy by 44% on average.

***Index Terms*—GPUs, Scheduling, Approximate Computing**

I. Introduction

Graphics Processing Unit (GPU)-based architectures are becoming the default accelerator choice for a large number of data-parallel applications ranging from high-performance computing (HPC) workloads to cryptographic applications. Because of their ability to provide high compute throughput at a competitive power budget, they are being employed into almost all kinds of computing systems, including many machines on Top500 [1] and Green500 lists [2]. One of the biggest impediments towards the continuous scaling of GPUs is the memory system energy consumption [3]. A large fraction of DRAM access energy is related to the fact that multiple high-energy consuming DRAM operations such as row activations and precharges must be performed, so as to access data from a DRAM row (page). These operations are required to ensure the data from the correct row is present in the row buffer, which is a limited-sized hardware structure attached to each DRAM bank. If accesses to the same row can be scheduled together without switching in and out the row buffer data (i.e., row buffer locality can be enhanced), they can incur much less row energy. Quantitatively, this energy can be around 25-50% of the total DRAM energy [4]–[7] and is dependent on the row buffer locality workloads (higher the row buffer locality, the lower the DRAM energy). Hence, it is preferred to reuse the buffered data of a row as much as possible to improve the row buffer locality and reduce the energy consumption.

We observe that several GPGPU applications suffer from poor row buffer reuse (also referred to as *row thrashing*). It can happen even with the popular First-Row First-Come-First-Serve (FR-FCFS) scheduler that leverages a large re-order pending request queue and an open-row policy which is typically employed to maximize the row buffer locality. This is not only caused by the GPU scheduling policies at the core but is also dependent on the applications' algorithms and their data placement mechanisms. Moreover, the multi-threading nature of the GPUs can cause severe contention and interleaving of requests at the memory controller, which can also lead to poor row buffer locality. To address this problem, we performed a detailed characterization of row buffer locality in GPUs and revealed two key insights. First, the current GPU memory scheduling policies are too aggressive in reducing latencies of requests: requests in the pending queue are issued to their destined DRAM banks as soon as these DRAM banks finish serving the previous requests. Second, the current memory scheduling policies are too strict in terms of fetching only the *exact* values from the DRAM banks. Therefore, an entire DRAM row has to be fetched into the row buffer even if it is poorly reused. We argue that these aggressive and strict policies are sub-optimal towards improving row buffer locality.

Our lazy memory scheduler relaxes the aforementioned constraints by leveraging the fact that several GPGPU applications are latency and error tolerant [8], [9]. Specifically, our proposed memory scheduler works in two modes: delayed and approximate. The delayed memory scheduling (DMS) carefully delays (i.e., increases the access latency) the issuing of both read and write pending memory accesses so that more requests can be accumulated in the FR-FCFS pending queue. This helps the memory scheduler to find more requests (i.e., will have more visibility) that can be co-scheduled back to back to the same DRAM row leading to improved row buffer locality. Because several GPGPU applications are inherently latency tolerant as they spawn thousands of threads to hide the long memory access latencies (which is not the case for most of the workloads executed on CPUs), we find that the additional delay does not affect performance significantly for many GPGPU applications. However, for certain applications that cannot tolerate latency significantly, DMS is also able to find an appropriate delay to avoid severe loss in performance.

The approximate memory scheduling (AMS) is based on our observation that a large portion of row activations is caused by only a small portion of memory accesses. To this end, the

goal of AMS is to find these accesses with low row buffer localities in the pending queue and return them immediately instead of issuing them to the DRAM banks. The values of such a small portion of memory accesses can then be approximated using various existing techniques [10]–[12] on their way back to the cores. These techniques bound the error with the help of programmer annotations and by predicting only a fraction of memory requests (called as prediction coverage). We demonstrated the effect of approximation on the application output by using a simple value predictor, which makes use of the readily available data in the associated L2 caches of the memory partitions. Because of the fact that many GPGPU applications are error tolerant or can accept limited losses in the output quality [8], [12], we find that such an approach can help in significantly reducing the number of row activations. Overall, AMS focuses on the problem of *when* to approximate and allow the new or existing works [10], [11] to address the equally important problem of *how* to approximate.

To the best of our knowledge, this is the first work that improves the row buffer locality and reduces row energy in GPUs via carefully delaying and/or approximating the memory requests (i.e., trading-off modest performance and application accuracy for better row buffer locality). In summary, this paper makes the following contributions.

- We demonstrate that delaying the scheduling of memory requests can significantly improve the overall row buffer locality because the memory controller can find more requests that can be scheduled back to back to the same row. Given that several GPGPU applications are latency tolerant, we do not observe notable performance reduction in such applications. To control the performance loss caused by delays, we devise a low-overhead dynamic mechanism that limits the delay by ensuring that utilization of DRAM stays above a certain threshold.
- We demonstrate that a small fraction of memory requests can cause a large fraction of row activations (i.e., there is non-uniform reuse of row buffers). Therefore, approximating a limited number of requests (bounded by the prediction coverage) can significantly reduce the row energy, without notably degrading the output quality of error-tolerant GPGPU applications. To improve the row buffer locality more effectively under a limited prediction coverage, we devise a low-overhead dynamic mechanism that is able to prioritize the approximation of requests with relatively low row buffer localities.
- Our newly proposed lazy memory scheduler for GPUs realizes the aforementioned contributions via delayed memory scheduling (DMS) and approximate memory scheduling (AMS), respectively. We show that DMS and AMS can work separately or together while improving the effectiveness of each other. Our evaluation shows that across a variety of GPGPU applications, row energy can be reduced by 12% using DMS, 33% using AMS, and 44% using a combination of both schemes. We achieve these results with less than 1% IPC loss, with an acceptable loss in application accuracy, and without requiring additional buffer space beyond what already exists in the baseline memory controllers.

II. Background and Metrics

A. Baseline GPU Architecture

GPUs achieve high throughput because they are capable of executing a large number of threads concurrently. We consider a generic GPU architecture consisting of several cores (known as Streaming Multi-processors (SM) in NVIDIA terminology), which are connected to memory partitions via an interconnect as shown in Figure 1. In order to support large amount of thread-level parallelism in GPUs, each SM consists of several processing elements (PEs), supported by a large register file (for saving context of a large number of concurrent threads so as to minimize context switch overhead) and all memory partitions manage high bandwidth memories (for fast data access to large number of concurrent threads). Each SM also has a private L1 cache and each memory partition is attached to a shared L2 cache. Each memory partition also has a memory controller (details will be discussed next) that is responsible to schedule L2 cache misses (i.e., memory requests sent from the L2 cache) to the DRAM. We evaluate the proposed techniques on a cycle-level GPU simulator – GPGPU-Sim [13] (Table I) and collect energy-related measurements using GPUWattch [14].

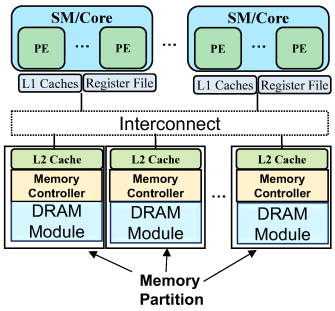


Fig. 1: Overview of our baseline GPU Architecture.

TABLE I: Key configuration parameters of the simulated GPU

SM Features	1400MHz core clock, 30 SMs, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32KB register file, Max. 1536 threads (48 warps, 32 threads/warp)
L1 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
L2 Cache	8-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 (MCs), FR-FCFS scheduling [15], 16 DRAM-banks/MC, 4 bank-groups/MC, 924 MHz memory clock, global linear 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
Interconnect	1 crossbar/direction (30 SMs, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

B. DRAM Organization and Operations

We provide a high-level description of DRAM organization/-operations and refer readers to the existing rich literature [4], [6], [7], [9], [16]–[18] on DRAM for more details.

DRAM organization. The data is spread across multiple channels (partitions) for achieving high memory bandwidth. For each channel, the memory operations are performed at the granularity of DRAM banks. Each bank consists of the cell arrays and a row buffer (sense amplifier) to read data from or write data to the cell arrays [19]. The cell arrays are where the

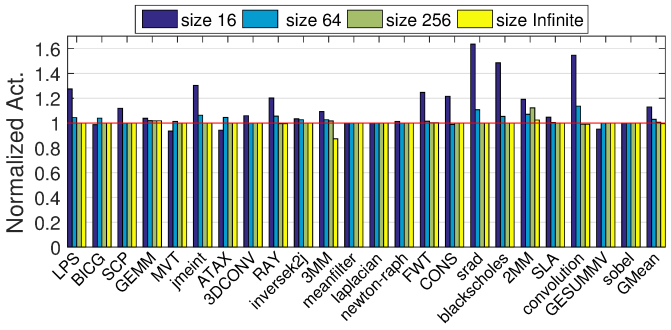


Fig. 2: Effect of pending queue size on the number of row activations (Act.). Results are normalized to the case of pending queue size 128.

data is stored and consist of many rows (pages) and columns (bits). Each memory channel is also associated with a memory controller, which buffers the pending memory requests in a request pending queue and determines the order to serve them in their destined banks.

DRAM operations. In order to serve a read or write request to a bank, a whole row in the cell array must first be activated (i.e., opened) to fetch its data into the row buffer. After the pending accesses to the current row are served and before the pending accesses for other rows can be served, the data present in the row buffer must be restored back to the cell arrays to safely keep the correct data values of the row. Finally, a precharge operation also needs to be performed in order to ensure that the next activation operation of a row can be performed successfully. The access energy is dependent on the access type. The row buffer hit request consumes less energy compared to the row buffer miss request. It is because serving the row buffer miss request involves costly operations such as activation, restore and precharge. The energy consumed by these operations (referred to as row energy in this paper) contributes significantly to the total DRAM energy consumption [6], especially when the row buffer locality is low (i.e., the ratio of row buffer miss requests is high among all DRAM requests). Note that although we use GDDR5 DRAM model as our baseline in this paper, the row locality concerns are pervasive across all memory technologies (e.g., HBM, HBM2) [6], [7].

C. Baseline Memory Controller

Our baseline memory scheduler is First-Row First-Come-First-Serve (FR-FCFS) [13], [15], [20], [21], which is commonly employed to optimize for row buffer locality in GPUs. Specifically, FR-FCFS prioritizes row buffer hit requests over other requests, including older ones. If no request is a row buffer hit, then FR-FCFS prioritizes older requests over younger ones. We also use the open-row policy together with the FR-FCFS scheduler to minimize the row activations. Both read and write requests are served as per FR-FCFS scheduling policy [13]. A large re-order pending request queue can potentially help in reducing the number of row activations by making more requests visible to the FR-FCFS memory scheduler. For a series of GPGPU applications, Figure 2 shows that the number of activations reduces (i.e., row buffer locality increases) with larger queue sizes. As the rate of decrease saturates after the size of 128, we use it as our baseline configuration.

D. Evaluation Metrics and Terminology

We summarize the definitions that will be used in this paper.

DRAM Locality-related Terminology. Row Buffer Locality (RBL) is defined as the number of requests that are scheduled back-to-back to the same DRAM row during the time it is activated in the row buffer. In this context, the notation RBL(X) would imply that X requests access the same row back-to-back before it is closed. The Average Row Buffer Locality (Avg-RBL) is defined as the ratio of the total number of memory requests to the total number of row activations. We also use the notation RBL(X - Y) to denote all the rows which have RBLs that belong to the range RBL(X) to RBL(Y).

Delay-related Terminology. We define Delay as the minimum number of required cycles spent by every request in the pending queue before it can be considered for scheduling. These required cycles are enforced by our proposed *delayed* memory scheduling (DMS) which will be introduced in the following sections. In this context, we use the notation DMS(X), where X indicates the minimum required cycles of delay, to denote the delay configuration of the pending queue. The largest value of X at which the application performance (in terms of Instructions-per-Cycle (IPC)) degrades no more than a user-defined percentage is defined as the Maximum Tolerable Delay (MTD). For our purposes, we tolerate up to 5% IPC degradation compared to the baseline but this number can also be changed by the user.

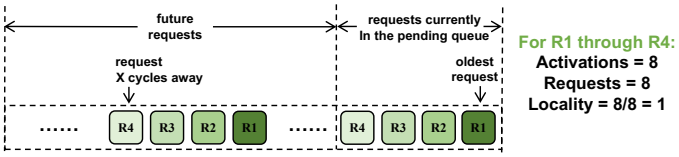
Approximation-related Terminology. The coverage is defined as the percentage of memory global read requests that are *not* served by the DRAM banks but instead *dropped* from the memory pending queue and returned immediately to the reply queue. It will then be recognized and *approximated* by the value predictor on its way back to the core. We consider these global read requests for approximation only when they are in rows with low RBLs. In this context, we define RBL-Threshold, Th_{RBL}, which is the value up to which the row is considered to have low RBL and hence those requests are the candidates for approximation. For example, if Th_{RBL} is equal to 3, it implies that all rows with RBL(1), RBL(2), and RBL(3) have low RBL. The dropping of requests in rows with low RBL is executed by our proposed *approximate* memory scheduling (AMS), which will be introduced in the following sections. We use AMS(Th_{RBL}) to denote the approximation configuration. The approximation conducted by AMS and value predictor can cause a certain level of output quality degradation, which we estimate with the application error. The application error is defined as the average relative error between the output of the baseline version of an application and the output of the same application with load value approximation. In general, higher coverage can lead to larger application error [12].

III. Motivation and Analysis

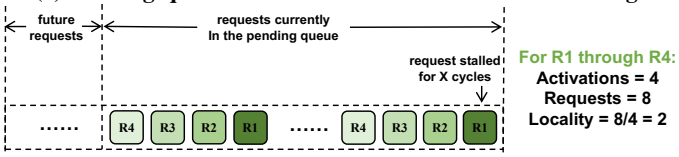
Our goal is to improve the average row buffer locality (i.e., Avg-RBL) by reducing the number of poorly reused rows. To this end, we propose two mechanisms: a) delayed memory scheduling (DMS), which trade-off scheduling delay (and potentially performance) for better Avg-RBL and b) approximate memory scheduling (AMS) which trade-off output quality for better Avg-RBL. In this section, we will motivate these trade-offs and show their effectiveness. We will also discuss how both these scheduling techniques can work together for even higher improvements in the Avg-RBL.

A. Delayed Memory Scheduling (DMS)

The baseline FR-FCFS scheduler attempts to schedule pending memory requests to the DRAM bank as soon as it is idle. Interestingly, we find that such timely scheduling of requests by the memory controllers actually disallows optimal reuse of data present in row buffers. To understand this observation, consider an illustrative example shown in Figure 3. The first scenario in Figure 3(a) depicts the baseline case of FR-FCFS scheduling. As shown in the figure, there are currently four pending requests in the memory controller's pending queue and these four requests belong to four different DRAM rows (R1, R2, R3, R4) of the same bank. Also, there are many more requests destined to the same bank but have not yet arrived at the pending request queue. Among such requests, there are four more requests that belong to the same four DRAM rows (R1, R2, R3, R4). For the baseline scheduler that timely issues all these requests, we find that the first four requests in the pending queue are issued back to back to the DRAM bank, leading to 4 activations for R1 through R4. When the remaining four requests arrive at the pending queue, four additional activations will also be required to serve them. Therefore, eight activations are required to serve all eight requests of R1 through R4, leading to an Avg-RBL of 1.



(a) Pending queue with the baseline FR-FCFS scheduling.



(b) Pending queue with DMS.

Fig. 3: An example illustrating the benefits of delayed memory scheduling due to increased visibility to the memory controller. Eight requests are shown in total destined to four DRAM rows (R1, R2, R3, R4).

In order to improve the Avg-RBL, we propose the *delayed* memory scheduling (DMS). DMS carefully delays the issuing of each pending memory request in the hope that more requests destined to the same row of a bank will show up in the pending queue. To illustrate this, consider the case as shown in Figure 3(b) where the issuing of all requests have been delayed for X cycles. Hence, by the time the other four requests have reached the pending queue, the first four requests to R1 through R4 are still in the pending queue. Therefore, only four activations are required to serve all eight requests, leading to an Avg-RBL of 2 (twice of the baseline case).

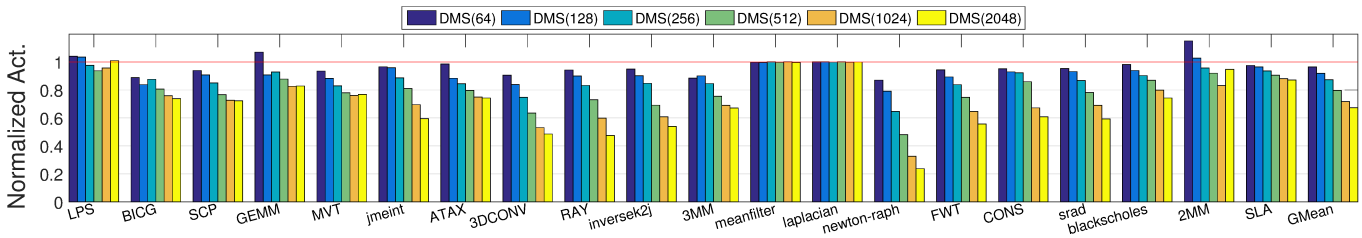
Figure 4(a) shows the normalized number of activations across a variety of GPGPU applications. For all of these applications, each of their requests (that does not lead to a row hit) is delayed by X cycles in the pending queue, denoted by DMS(X), before it can be served by a DRAM bank (more details are explained in Section IV). We show the results for when X is equal to 64, 128, 256, 512, 1024, and 2048 cycles. We find that many applications are sensitive to delay – the higher the delay, the higher the chance of finding requests destined to the same rows, which leads to fewer row activations. On average, the activation reduction can be as high as 31%, when a delay of 2048 cycles is used. Figure 5 shows the distribution of row activations based on their RBLs with different delays for two applications. As we observe, for both applications, the proportion of row activations with RBL(1) (i.e., only one request accesses the activated row before it is closed – Section II-D) reduces significantly with the increase of delay. Meanwhile, the proportions of row activations with higher RBLs have increased. This shift in the RBL of row activations effectively shows how DMS can help to improve the Avg-RBL for real applications.

On the negative side, the increase in delay can degrade the overall performance. Thanks to the latency tolerance of GPGPU applications, the increase of delay has a limited impact on the performance as shown in Figure 4(b). Many applications retain their baseline performance up to 95% even at very large delays (e.g., 1024 cycles). However, IPC's sensitivity to delay varies for different applications and hence it is critical to determine an appropriate value of delay to carefully trade-off the activation reduction with the performance.

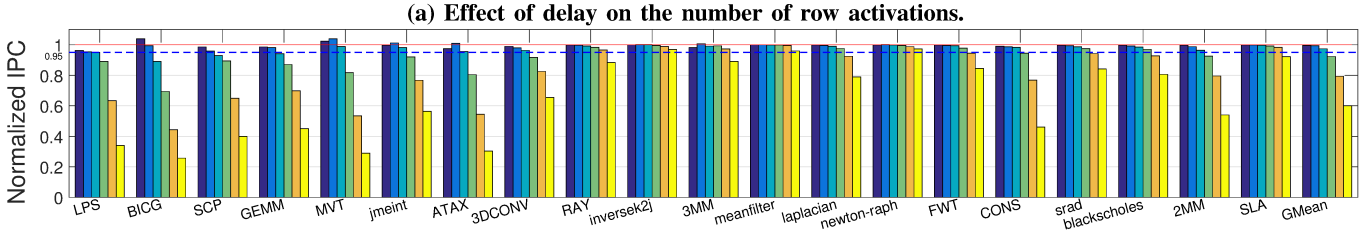
B. Approximate Memory Scheduling (AMS)

In order to further improve the Avg-RBL, we determine which pending requests have low RBLs and propose to return these requests immediately instead of issuing them to the DRAM banks. Subsequently, their values are approximated using existing techniques on their way back to the cores. Our proposal is motivated by the observation that for many GPGPU applications, a small portion of memory requests contributes to a high proportion of total row activations. The cause of this is multi-fold as it depends not only on the applications' algorithms and data placement mechanisms but also on the runtime behaviors driven by the warp or thread-block scheduling techniques. Nevertheless, as we will discuss further, our proposed techniques are also complementary to other optimizations that may improve Avg-RBL separately.

AMS works on row activations that only contain memory read accesses, as memory write accesses are typically not the

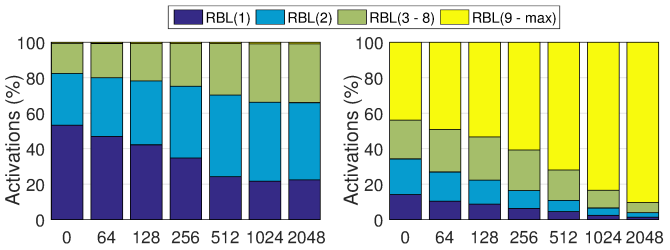


(a) Effect of delay on the number of row activations.



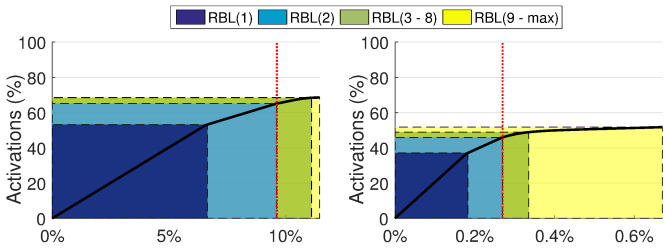
(b) Effect of delay on performance (IPC).

Fig. 4: Effect of delayed memory scheduling on the number of activations and performance. Results are normalized to the baseline architecture (Section II), which does not employ delayed or approximate scheduling.



(a) SCP **(b)** 3DCONV

Fig. 5: Effect of delayed memory scheduling on activation proportions of each RBL. x-axis indicates delay. y-axis indicates each component's proportion to the total number of activations.



(a) CDF for GEMM **(b) CDF for** 3MM

Fig. 6: The cumulative distribution of total row activations for requests associated with different RBLs. x-axis is the proportion of requests sorted by their RBLs.

targets for value approximation techniques. Figure 6 shows the proportion of row activations from the rows that are opened to serve only global read requests. We sort these requests in increasing order of their associated row activations' RBLs. Note that the x-axis denotes the proportion to the total number of requests. The y-axis denotes the proportion to the total number of activations. The shaded regions on the curve indicate the portions contributed by each RBL category. As shown in Figure 6(a), for GEMM around 10% of memory read requests associated with RBL(1) and RBL(2) cause about 65% of the total row activations. Similarly, as shown in Figure 6(b), for 3MM around 0.2% of memory read requests associated with RBL(1) and RBL(2) cause about 45% of the total row activations. This implies that a large fraction of row activations is caused by only a small fraction of memory requests.

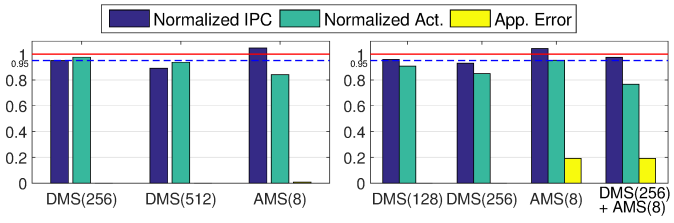
In order to leverage this observation to further reduce row activations, we propose *approximate* memory scheduling (i.e., AMS). AMS first recognizes the pending read requests which are not destined to the same rows as any of the pending write requests. Then AMS decides if these requests are associated with low-RBL row activations, which means that the RBLs that these requests are expected to bring are no greater than a specific RBL-Threshold (i.e., Th_{RBL}, more details in Section IV). Subsequently, AMS returns these requests immediately without issuing them to the DRAM banks. Finally, the values of such requests will be provided by a value approximation technique on their way back to the cores. We denote this as AMS(Th_{RBL}). Such an approach eliminates these low-RBL row activations in the DRAM banks, thereby significantly improving the Avg-RBL and reducing the DRAM energy. On the negative side, such an approach can lead to application-level error, which needs to be acceptable to the user. To control the application-level error, the number of approximated requests (i.e., prediction coverage) needs to be limited. Thus, within the coverage limit, finding the row activations with relatively low RBLs among all the activations is the goal of AMS. Further details of AMS are in Section IV.

C. Delayed and Approximate Scheduling

Having discussed the benefits of DMS and AMS separately, we now discuss how both DMS and AMS can work together to provide further benefits in terms of reducing the number of row activations and improving the performance. In this context, we consider the following two questions:

1) How can approximate memory scheduling help delayed memory scheduling: We find that AMS can help DMS especially for applications that belong to two categories:

Case 1. *The application's number of row activations is not sensitive to the change of delay.* For example, Figure 7(a) shows the normalized IPC and the normalized number of row activations for application LPS under three different cases. LPS has only a limited activation reduction (i.e., 2%) with its maximum tolerable delay (MTD) of 256 cycles. However, with a delay value of 512 cycles, LPS can reach its highest activation reduction (i.e., 6%), but also at the price of an IPC loss of 11%. On the contrary, if AMS is applied instead and approximates the requests associated with RBL(1-8) row activations (i.e., AMS(8)), LPS can get 16% activation reduction and 5% IPC improvement, only at the cost of less than 1% application error which is a minimal quality loss. Therefore, AMS is useful when DMS cannot effectively reduce the number of row activations as shown in this case.



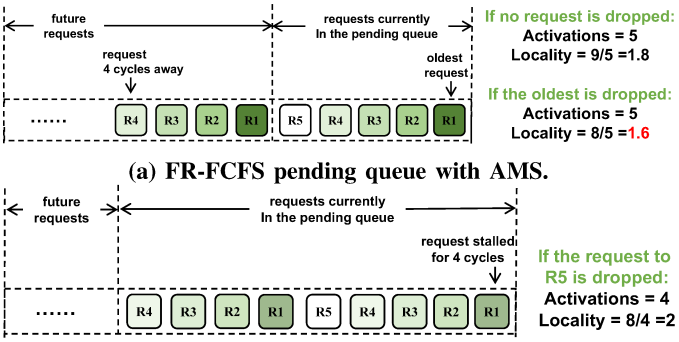
(a) LPS **(b)** SCP

Fig. 7: Examples illustrating how approximate memory scheduling can help delayed memory scheduling.

Case 2. *The application's number of row activations is sensitive to the change of delay, but the performance loss is preventing DMS from adopting higher delay values.* For example, Figure 7(b) shows different metrics for application SCP under four different cases. With DMS(128), the activation reduction can reach 9% at the cost of a 4% IPC loss. As the value of delay increases from 128 to 256, the activation reduction can further reach 15% at the cost of a 7% IPC loss. However, if we required that the performance loss must be under 5%, then DMS(256) should not be adopted and the further activation reduction cannot be achieved.

On the other hand, when applying AMS alone (the results of AMS(8) as shown in Figure 7(b)), the number of row activations reduces and also the IPC increases at the cost of increased application error. However, if we combine both DMS and AMS together (the results of DMS(256) + AMS(8) as shown in Figure 7(b)), SCP can adopt DMS(256) to obtain more activation reduction and still achieve less than 5% IPC loss. This means that the increase of IPC provided by AMS can compensate for the IPC loss caused by DMS. As a result, the value of delay can be further increased to obtain more activation reduction from DMS. In addition, AMS is able to work synergistically with DMS to further reduce the number of row activations, leading to a higher activation reduction. Therefore, AMS is useful to help increase the delay value in DMS as shown in this case.

2) How can delayed memory scheduling help approximate memory scheduling: We find that DMS can also help AMS in terms of activation reduction, as delaying the issuing of pending requests can help to more accurately identify the low-RBL row activations. To illustrate this, consider Figure 8, which shows that 9 requests are destined across 5 rows (i.e., R1 through R5) of the same DRAM bank and AMS is trying to find a request associated with an RBL(1) row activation to drop. Figure 8(a) shows a case when AMS is applied alone and there are 4 more requests destined to R1 through R4 of the same bank that have not yet reached the pending queue. Also, the time required for the bank to serve a request is sufficient for these 4 future requests to reach the queue. Since that the memory scheduler only has visibility of the requests currently in the pending queue, it observes 5 RBL(1) row activations at this point. Therefore, if AMS were to choose a request to be dropped, it would drop the first R1 as it is the oldest pending request. However, this would lead to even an Avg-RBL decrease from 1.8 (9/5) to 1.6 (8/5). This is because the total number of activations for these 9 requests is still 5, but the total number of requests is reduced from 9 to 8 (the first R1 is dropped). AMS cannot accurately drop R5 because the row indexes of future requests are unknown.



(a) FR-FCFS pending queue with AMS.

(b) FR-FCFS pending queue with DMS+AMS.

Fig. 8: Example illustrating how delayed memory scheduling (DMS) can help approximate memory scheduling (AMS) by comparing different schemes.

On the other hand, Figure 8(b) shows the case when AMS is applied together with DMS. As a result of the added delay by DMS, AMS will correctly drop R5 as it can observe now that only R5 has an RBL(1) row activation. Hence, the total number of activations is reduced from 5 to 4, and the total number of requests is reduced from 9 to 8, leading to an Avg-RBL increase from 1.8 (9/5) to 2 (8/4). In this case, AMS can more accurately identify low-RBL row activations as more requests are visible in the pending queue on account of applying DMS.

In summary, we find that both DMS and AMS can provide significant benefits in terms of activation reduction. Furthermore, they can also improve the efficiency of each other when applied together. In the next section, we will provide implementation details for both memory scheduling techniques.

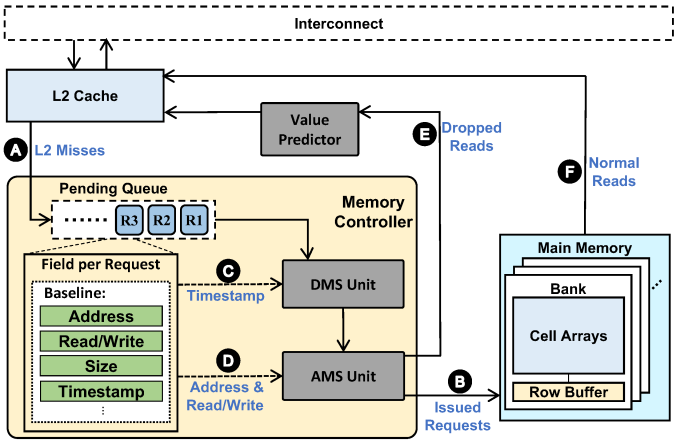


Fig. 9: Design overview of the lazy memory scheduler and associated components.

IV. Design and Operation

A. Overview

Figure 9 shows a high-level overview of our design. The L2 misses Ⓐ are buffered at the pending queue after they arrive at the memory controller. These pending requests are then issued to the DRAM banks following the FR-FCFS scheduling policy Ⓑ as soon as their destined DRAM banks become available (Section II). Our proposal focuses on seamlessly integrating our new memory scheduling schemes: DMS and AMS with the baseline FR-FCFS scheduler. In this context, Figure 9 shows three major components (shaded in gray color) of the lazy memory scheduler: delayed memory scheduling unit (DMS unit), approximate memory scheduling unit (AMS unit), and value prediction unit (VP unit). The DMS and AMS units coordinate with the memory controller to decide *which* and *when* the requests should be issued to the DRAM banks. The AMS unit also coordinates with the VP unit to decide *which* and *how* the requests will be approximated. Consequently, these units decide the sequence of row activations of DRAM banks so as to maximize the Avg-RBL.

The DMS unit can either work independently or with the AMS unit. In the former case, before opening a new DRAM row, the DMS unit checks whether the oldest request has spent at least X cycles (i.e., DMS(X)) in the pending queue. If true, then this oldest request is issued to the memory banks Ⓑ and its corresponding DRAM row is opened. The other pending requests destined to the same row are also issued back to back (regardless of their ages) as per FR-FCFS policy. To keep track of the delayed cycles per request, each request is assigned with a time stamp when it enters the pending queue. This time stamp is used by the DMS unit to check against the current time to get the value of delay Ⓒ (more details are in Section IV-B).

In the latter case, the DMS unit also checks whether the oldest request has spent at least X cycles in the pending queue. If true, it then checks the current prediction coverage, Th_{RBL}, and the pending requests' information Ⓓ to decide if this request should be dropped (more details are in Section IV-C). If all criteria are satisfied, the AMS unit will drop the request from the pending queue and send a dropped read signal Ⓔ to the VP unit to generate an approximate value. Otherwise, if the criteria are not satisfied, the request is issued to the memory banks Ⓑ, and the L2 cache is filled with accurate data served by the memory banks Ⓕ (the same as the baseline case).

B. Delayed Memory Scheduling Schemes

As discussed earlier in Section III, finding an appropriate value for delay is important for DMS. Higher values of delay would create more opportunities for the memory scheduler to improve the Avg-RBL, however, at the possible loss of performance. In this context, we propose two schemes: Static-DMS and Dyn-DMS, which calculates the value of X statically and dynamically, respectively.

Static-DMS: Static Delayed Memory Scheduling. The Static-DMS uses a delay of 128 cycles (i.e., DMS(128)), based on our empirical evaluations. As shown in Figure 4, 128 cycles is the maximum delay that can lead to less than 5% IPC losses across all tested applications. However, this static value of delay misses out on the opportunity of improving Avg-RBLs in applications with higher latency tolerances. It may also lead to more than 5% IPC losses in untested applications. Therefore, we further propose a scheme that dynamically decides the value of delay based on the latency tolerance of an application.

Dyn-DMS: Dynamic Delayed Memory Scheduling. We propose a profiling-based dynamic scheme, which is based on the fact that the performance degradation can be tracked locally at the memory controller via observing the bandwidth utilization (BWUTIL). For all the applications we used, we tested their BWUTILs and IPCs with different values of delay. As shown in Figure 10, their BWUTILs and IPCs are linearly correlated, which is also confirmed in previous works [22], [23]. For this reason, we can track the changes in DRAM bandwidth utilization locally at the memory controller to keep track of the changes in the overall performance.

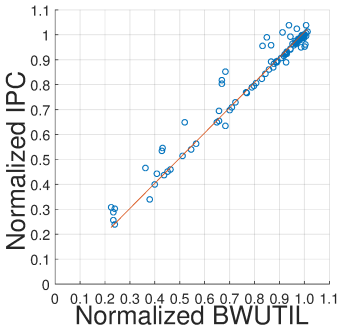


Fig. 10: Illustrating the relationship between IPC and BWUTIL.

Our Dyn-DMS mechanism is an iterative mechanism that attempts to find the maximum value of delay such that performance (reflected by bandwidth utilization) does not drop significantly (our threshold is 5%) compared to the baseline no-delay scenario. Dyn-DMS first samples the baseline BWUTIL for a window of 4096 memory cycles.[1] Note that in order to accurately sample the baseline BWUTIL, the co-running AMS scheme is temporarily halted during this window when applying DMS and AMS together. Then starting from a delay

[1]Based on our experiments, 4096 cycles is a suitable window size. An overly large window does not timely reflect the current BWUTIL, meanwhile, an overly small window is too sensitive to local spikes in BWUTIL (or coverage).

value of 128 cycles, the DMS unit gradually increases the value of delay (X) for the following 4096-cycle windows in steps of 128 delay cycles. At a particular delay, if the BWUTIL of that window starts to drop below 95% of the baseline, the iterative method stops and set the delay to be the last value that leads to a BWUTIL more than 95% of the baseline. This delay value X is also recorded. To capture the phases changes within an application, we restart the process after every 32 windows, however, we set the previously recorded delay value of X as the starting point for this iterative procedure to quickly settle to the optimal value. Note that the maximum value of X we use is 2048 and the minimum is 0 (baseline case).

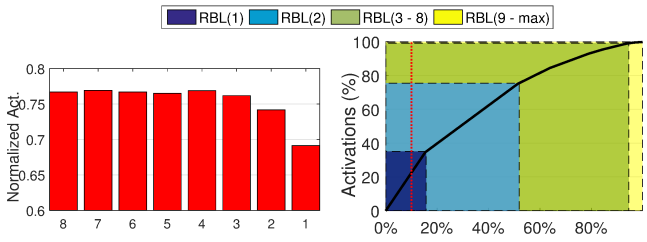
C. Approximate Memory Scheduling Schemes

As discussed earlier in Section III, with a coverage limitation, finding and dropping requests associated with relatively low RBLs are more favorable to reduce the number of activations. Therefore, an appropriate value for Th_{RBL} is important for AMS(Th_{RBL}). High values of Th_{RBL} would lead to unnecessarily approximating requests associated with high RBLs and wasting the limited prediction coverage. On the other hand, low values of Th_{RBL} may not provide enough opportunities to approximate if there are not enough requests associated with low RBLs, thereby limiting the potentials of Avg-RBL improvements.

The working procedure of the AMS unit has multiple steps. As using approximate value for critical data (e.g., pointers) may cause fatal errors for applications, we use *pragma* to annotate the approximable regions of data to guarantee the safety of applying value approximation. Hence, the AMS unit will only proceed if it detects that the oldest pending request is approximable. Second, the AMS unit verifies if the oldest request satisfies the delay criteria determined by DMS. Third, if the first criterion is satisfied, the AMS unit calculates the coverage based on the total number of requests dropped and the total number of requests received so far. It then checks if the coverage is less than the user-defined coverage value (we use 10%). Fourth, if the second criterion is also satisfied, the AMS unit iterates through the pending queue to obtain the RBL value associated with the request and checks if it is less or equal to Th_{RBL}. Also, during this iteration, the AMS unit ensures that all the other requests destined to the same row are global read requests, as we only approximate load values. If the fourth criterion is also satisfied, then this request will be dropped from the pending queue, instead of being issued to the memory bank. In addition, all other pending requests destined to the same row will also be dropped sequentially in the following memory cycles. If any of these three steps are not successful, as default, the request will be issued to the memory banks following the FR-FCFS policy.

We propose two schemes to realize the above goals and procedures: Static-AMS and Dyn-AMS, which calculates the value of Th_{RBL} statically and dynamically, respectively.

Static-AMS: Static Approximate Memory Scheduling. Based on our empirical evaluations, we found that the Th_{RBL} value of 8 is appropriate as it does not allow unnecessary approximations for requests associated with very high RBLs and at the same time provides enough prediction coverage for many applications. Therefore, AMS(8) is used for Static-AMS scheme. However, as different applications have very different RBL distributions, a static Th_{RBL} can be sub-optimal for some of the applications. For example, if the Th_{RBL} is too high for them, AMS cannot accurately target requests associated with lower RBLs under a limited prediction coverage. On the other hand, if the Th_{RBL} is too low for them, AMS cannot effectively reduce the number of activations because there are not enough requests for it to approximate. Therefore, there is a need to dynamically modulate the value of Th_{RBL} so as to more accurately target the low-RBL row activations while also maintaining the user-defined coverage (10%).



(a) The effect on activation reduction by reducing Th_{RBL} (x-axis) for SCP.

(b) CDF for SCP's activations. x-axis is request percentage of read-only rows sorted by RBL.

Fig. 11: Effect of reducing Th_{RBL}.

Dyn-AMS: Dynamic Approximate Memory Scheduling. For some applications, the Static-AMS (i.e., AMS(8)) may be suboptimal. For example, as shown in Figure 11(a), application SCP's number of activations can be further reduced when Th_{RBL} is reduced from 8 to 1. The reason for this can be explained with Figure 11(b). As shown in the Figure, most of the requests within the Th_{RBL} of 8 are associated with RBL(2 - 8). However, there are already more than 10% of the total requests associated with RBL(1) (i.e., the portion on the left of the red dashed line). Therefore, a Th_{RBL} value of 1 is most beneficial, as approximating 10% requests associated with RBL(1) leads to the highest activation reduction. Hence, dynamically modulating the Th_{RBL} is necessary to further improve the activation reduction for applications like SCP.

Based on this observation, we designed a profiling-based Dyn-AMS scheme. Similar to the Dyn-DMS, the Dyn-AMS is also an iterative approach that attempts to find the lowest value of Th_{RBL} such that the prediction coverage does not drop below the user-defined value. Note that we empirically use 10% coverage throughout the paper and the Th_{RBL} range we use in the Dyn-AMS is 1 to 8. The AMS unit starts with the Th_{RBL} value of 8 and samples the coverages for consecutive windows of 4096 memory cycles. First, as long as the coverage can achieve the user-defined coverage, the AMS unit gradually decreases the Th_{RBL} value in steps of 1 in consecutive 4096-cycle windows. Second, once the coverage goes below the user-defined coverage in a window, the AMS unit gradually increases the Th_{RBL} value in steps of 1 until the coverage returns to the

TABLE II: List of evaluated GPGPU applications. See Table III for more details.

Abbr.	Description	Input	Group	Thrashing Level	Delay Related		Approximation Related	
					Delay Tol.	Act. Sens.	Th_{RBL} Sens.	Err. Tol.
RAY [13]	Ray Tracing	Matrix	3	High	High	High	Low	High
inversek2j [24]	Inverse kinematics for 2-joint arm	Coordinates	3	High	High	High	Low	High
newtonraph [24]	Equation solver	Image	4	High	High	High	Low	Low
FWT [13]	Fast Walsh Tranform	Matrix	4	High	Medium	High	High	Low
MVT [25]	Matrix Vector Product and Transpose	Matrix	2	High	Medium	High	Low	High
jmeint [24]	Triangle intersection detection	Coordinates	2	High	Medium	High	Low	Medium
ATAX [25]	Matrix Transpose, Vector Multiplication	Matrix	4	High	Medium	High	Low	Low
3DCONV [25]	3D Convolution	Matrix	2	High	Medium	High	Low	Medium
CONS [25]	1D Convolution	Matrix	4	High	Medium	High	Low	Low
srad [13]	Speckle Reducing Anisotropic Diffusion	Image	4	High	Medium	High	Low	Low
LPS [13]	3D Laplace Solver	Matrix	1	High	Medium	Low	High	High
BICG [25]	BiCGStab Linear Solver	Matrix	1	High	Low	High	High	Medium
SCP [13]	scalar products	Matrix	1	High	Low	High	High	Medium
GEMM [25]	Matrix Multiplication	Matrices	4	High	Low	Medium	High	Low
blackscholes [24]	Black-Scholes Option Pricing	Matrix	4	Medium	Medium	High	High	Low
2MM [25]	2 Matrix Multiplications	Matrices	4	Medium	Medium	Medium	Low	Low
3MM [25]	3 Matrix Multiplications	Matrices	3	Low	High	High	Low	High
SLA [13]	Scan of Large Arrays	Matrix	4	Low	High	Medium	Low	Low
meanfilter [24]	Convolution Filter for Noise Reduction	Image	3	Low	High	Low	Low	High
laplacian [24]	Image sharpening filter	Images	3	Low	Medium	Low	Low	Medium

TABLE III: Application features and intensity classifications. The thresholds are used only to facilitate the discussion in Section V.

Feature	Description	Categories (by X Range)		
		Low	Medium	High
Thrashing Level	The application has $X\%$ requests in rows with RBL(1 - 8).	$[0,3)$	$[3,10)$	$[10,100)$
Delay Tolerance	The application has a MTD of X.	$[0,256)$	$[256,1024)$	$[1024,+\infty)$
Activation Sensitivity	The application's activation reduction is $X\%$ compared to the baseline when 2048 cycles delay is applied to the FR-FCFS pending queue.	$[0,10)$	$[10,20)$	$[20,100)$
Th_{RBL} Sensitivity	The application's maximum activation reduction is $X\%$ compared to the baseline when reducing its Th_{RBL} from 8 to lower values.	$[0,5)$	NA	$[5,100)$
Error Tolerance	The application shows $X\%$ application error when using our proposed value approximation technique (Section IV-D) at 10% coverage or its maximum available coverage less than 10%.	$[20,+\infty)$	$[5,20)$	$[0,5)$

user-defined coverage again in consecutive 4096-cycle windows. These steps are repeated until the end of application execution.

D. Value Prediction Unit

The Value Prediction Unit (VP unit) is responsible for approximating the values of requests that are dropped by the AMS unit. Since the VP unit works independently and is orthogonal to the memory scheduling schemes, we can support a large variety of previously proposed value prediction mechanisms such as [10]–[12], [26]. Similar to prior works, AMS uses programmer annotations to bound the approximation errors as the criticality of instructions presumably could only be identified by the programmer [27]–[31]. AMS requires the following information from the programmer, as shown in the example of Listing 1: a) the approximable loads which are error tolerant, and b) the prediction coverage which limits the total number of approximations.

```
#pragma pred_coverage{10%}
#pragma pred_var{B}
C[i] = A[i] + B[i];
```

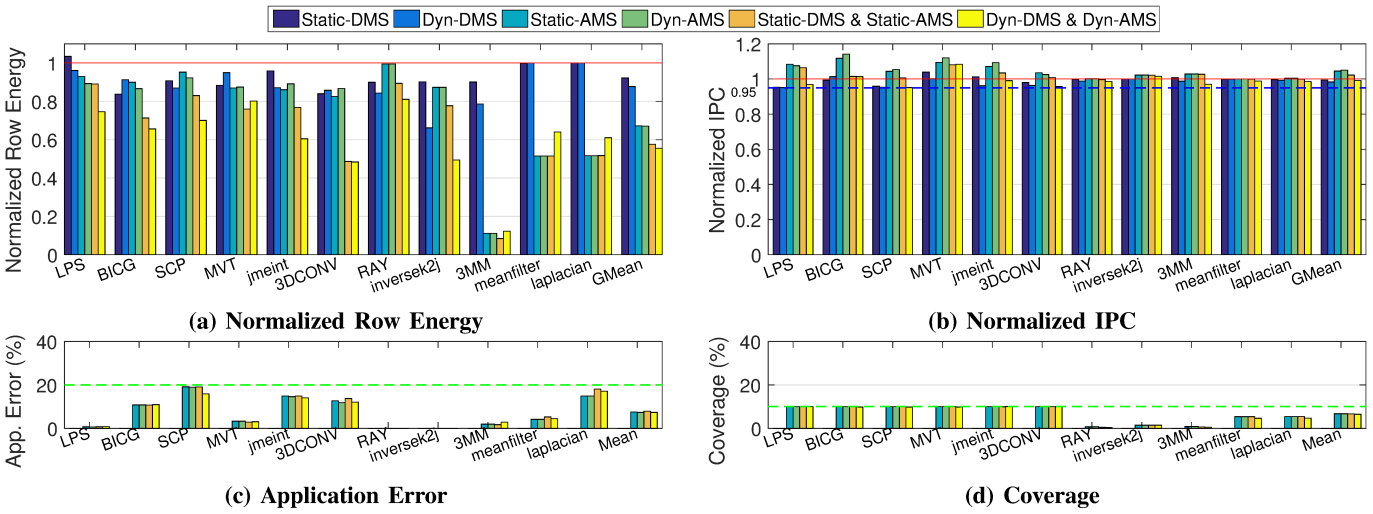
Listing 1: Example of Code Annotation

To demonstrate how AMS works, we designed a simple but effective VP unit that is based on the intuition that nearby addresses may store similar values and hence the value of a cache line can be approximated by a nearby cache line with limited error [10]. In order to predict the values for the dropped requests, we search in the nearby cache sets of the L2 cache and use the values from cache lines with nearest addresses as their approximate values.[2] To minimize the searching overhead, we carefully choose the search radius of nearby sets and take advantage of the existing associative search hardware to search in the cache ways of a set. We find that the searching overhead is negligible compared to the performance improvement introduced by value approximation. We will discuss the performance and output quality results in Section V. Note that we first warm up the L2 cache with a sufficient number of requests to prepared for the searches, and thus AMS is initially disabled until the cache is ready.

E. Hardware Overhead

The DMS unit requires one comparator and one adder to do comparisons for the functionalities of DMS. One 16-bit counter is required for Static-DMS and Dyn-DMS to store the current delay value of X. For Dyn-DMS, the DMS unit requires one 32-bit counter to store the baseline BWUTIL, one 32-bit counter to store the current BWUTIL, one 16-bit counter to store the cycles during profiling, one 8-bit counter to store the number of windows during profiling. The AMS unit requires one multiplier, one adder and one comparator for the

[2]In this simple model, we did not consider the error propagation caused by the reuse of approximated cache lines. However, we have tested with a more advanced model (that considers reuse) and have observed similar application error results.



(a) Normalized Row Energy **(b) Normalized IPC**

(c) Application Error **(d) Coverage**

Fig. 12: Comparison of different schemes with different metrics for applications with Medium or High Error Tolerance. Row Energy and IPC results are normalized to the baseline that does not adopt DMS or AMS.

operations of AMS. Static-AMS and Dyn-AMS require 1 bit to store the read/write condition and 1 bit to store the current memory space condition for the row of the oldest request, two 64-bit counters to store the total number of requests and the total number of approximated requests for calculating coverage, one 8-bit counter to store the RBL of the current request's row, one 8-bit counter to store the current Th_{RBL}, one 32-bit counter to store the index of the dropped request's row. For Dyn-AMS, the AMS unit requires one 16-bit counter to store the cycles during profiling.The VP unit requires nine adders, one MUX, one comparator for searching the nearest cache line, one 8-bit counter to store the radius, one 64-bit counter to store the tag of the dropped read request, two 64-bit counters to store the minimal tag distance and its corresponding address. Overall, the lazy memory scheduler requires 1 multiplier, 11 adders, 1 MUX, 3 comparators and 498 bits of buffer space in addition to the baseline memory controller. We believe this hardware overhead is modest in comparison to the energy savings provided by DMS and AMS. Finally, our mechanisms do not require any modification to the existing DRAM protocols.

V. Experimental Results

We evaluate our lazy memory scheduling techniques on a wide range of applications described in Table II. The applications are selected so as to cover all important features that are relevant to our schemes. We list these features and their intensity classifications (e.g., Low, Medium, High) in Table III. We use annotations to make sure that we only approximate global read requests which do not contain pointers or lead to fatal errors so that value approximation can be applied to all applications safely. For the ease of presenting results in this section, we group these applications into 4 different groups:

Group-1: These applications have high or medium error tolerance and also show high Th_{RBL} sensitivity. Therefore, both AMS and DMS can be applied and likely to benefit.

Group-2: These applications have high or medium error tolerance, thus the AMS related schemes can be applied. However, they show low Th_{RBL} sensitivity, so Dyn-AMS may not show clear benefits in terms of activation reduction.

Group-3: These applications have high or medium error tolerance, thus the AMS related schemes can be applied. However, since they either have very few requests associated with RBL(1 - 8) (Low Thrashing Level), or have very limited rows that are only accessed by read requests when opened, their coverages cannot reach 10%.

Group-4: These applications have low error tolerance and thus the AMS related schemes should not be applied. However, for these applications, the DMS schemes can still be applied for reducing the number of row activations.

Effect on Row Energy. Figure 12(a) shows the normalized row energy across all schemes. We make four observations. First, overall the Static-DMS and Dyn-DMS are able to reduce row energies by 8% and 12%, respectively. Second, overall the Static-AMS is able to reduce 33% of row energy, which is more than that of the Static-DMS schemes. The Dyn-AMS does not show improvement over the Static-AMS for Group-2 and Group-3 applications. However, Group-1 applications overall show 7% row energy reduction in the Static-AMS and 11% in the Dyn-AMS. Third, for Group-1 and Group-2 applications, when combining Static-DMS and Static-AMS together, their average row energy reduces by 27%. This is 7% more than when Static-DMS and Static-AMS are applied separately. When combining Dyn-DMS and Dyn-AMS together, it shows the largest row energy reduction of 34%. This reduction is 7% more than when applying Static-DMS and Static-AMS together, and is 13% more than the total reduction of when applying Dyn-DMS and Dyn-AMS separately. Finally, when applying Dyn-AMS together with Dyn-DMS, Group-1, Group-2 and Group-3 applications overall achieve 44% row energy reduction. However, a few Group-3 applications (i.e., `3MM`, `meanfilter`, `laplacian`) show less row energy reduction

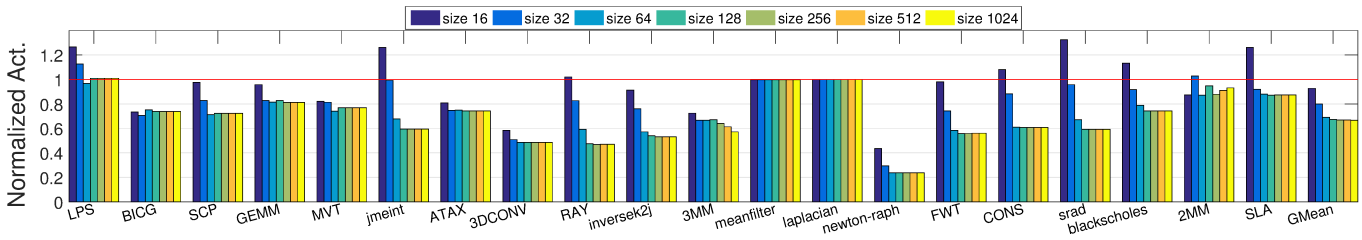
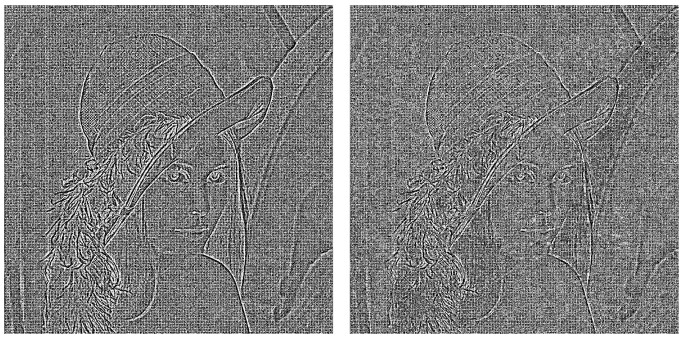


Fig. 13: Effect of pending queue size on the number of activations (normalized to the baseline) with DMS(2048).

than the other AMS related schemes. This is due to a small coverage decrease as shown in Figure 12(d) because Group-3 applications already have limited coverage and the profiling of Dyn-DMS reduces the number of requests dropped by Dyn-AMS (Section IV-B).

Effect on Memory Energy and Peak Bandwidth. The lazy memory scheduler's benefit in row energy reduction is caused by the improvement of the application's Avg-RBL. Therefore, it is independent of the memory technology used as long as it adopts similar structures as the row buffer. However, system-wise, its energy reduction is dependent on the memory technology. For example, if we apply Dyn-DMS and Dyn-AMS together on HBM1 where row energy constitutes nearly 50% of the memory system energy [6], we observe on average 22% memory system energy reduction with our tested applications. Similarly, for HBM2 where row energy can constitute 25% of its total energy, we observe on average 11% memory system energy reduction. Traditionally, the overall power budget of a high-end GPU card is limited to around 300W and its memory power budget is generally capped at 60W when operating at peak bandwidth. [7]. Therefore, in terms of absolute savings with HBM2, the lazy memory scheduler can achieve: a) up to 8W memory power reduction while achieving the same peak bandwidth or b) up to 90 GB/sec higher peak bandwidth under the same 60W memory power budget.

Effect on Performance. Figure 12(b) shows the changes in IPC across all schemes. Overall, we find that all our schemes do not lose more than 5% IPC. We make three observations. First, the Static-DMS and Dyn-DMS show larger IPC losses because of the additional delay. Also, the IPC of Dyn-DMS can approach closer to the 95% threshold, resulting in more row energy reductions. Second, the Static-AMS and Dyn-AMS show IPC improvement. Specifically, overall Dyn-AMS shows more improvement than Static-AMS, indicating that it can improve more performance by potentially dropping the requests in rows with lower RBLs. Finally, when combining Static-DMS and Static-AMS together, overall the IPC improves by 2%. When combining Dyn-DMS and Dyn-AMS together, overall the IPC loss is less than 1%. Both cases show higher IPC than the Static-DMS or Dyn-DMS scheme, because of the usage of AMS. We conclude that all our schemes are able to effectively restrict the IPC loss to be less than 5%. Specifically, AMS can help to compensate for the IPC loss caused by DMS. The combination of DMS and AMS can provide a good trade-off between row energy reduction and performance loss.

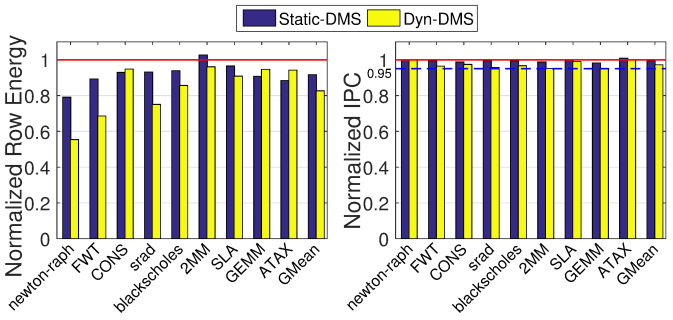


(a) Accurate Output **(b) Approximate Output**

Fig. 14: Comparison between the accurate and the approximate output (which has 17% Application Error and is generated when the Dyn-DMS and Dyn-AMS schemes are applied together) for application `laplacian`.

Effect on Application Error. Figure 12(c) shows application errors across all schemes. Note that the application error for the Static-DMS and Dyn-DMS are all zeros because no approximation is applied. We find that with our VP unit design, different applications show different application errors, meanwhile for each application, there are only small differences of application error with similar prediction coverages (Figure 12(d)) across different schemes. With the 10% coverage limitation, the average application error is 7% for all the AMS related schemes. Figure 14 shows the image output of application `laplacian` for the accurate baseline case and the Dyn-DMS and Dyn-AMS combination case. We observe that with 17% application error, the image shows a limited level of quality degradation. We conclude that under our VP unit design, limiting the coverage is an effective way to limit the application error. Moreover, value approximation is a feasible way to reduce row energy and improve performance as many applications can tolerate certain levels of error and are suitable for applying the AMS schemes. We also expect to see significant application error reduction if the AMS related schemes are applied together with the previously proposed value prediction techniques [10]–[12], [26] because they are more sophisticated and have shown much less application output quality loss when working with the same 10% coverage limitation.

Effect of FR-FCFS Pending Queue Size. When applying DMS, more requests are likely to be piled up in the pending queue, increasing the possibility to find row hits. However, if the pending queue is frequently full, future requests may often



(a) Normalized Row Energy **(b) Normalized IPC**

Fig. 15: Comparison of different schemes in the delay-only mode for applications with Low Error Tolerance.

be blocked from entering it, limiting the Avg-RBL improvement of DMS. Therefore, it is important that the pending queue size is sufficient to support the increased pending requests in DMS. Figure 13 shows the effect on the number of row activations when using different pending queue sizes with the maximum allowed delay (i.e. DMS(2048)). And starting from size 128 the activation numbers for all applications tend to be stable. We conclude that a pending queue size of 128 (i.e., the baseline size) is sufficient to apply DMS.

Delay-Only Mode for Low Error Tolerance Applications. For applications with low error tolerance, even if AMS cannot be applied, we can still use DMS to reduce their row energy. Figure 15(a) and (b) show normalized row energy and IPC, respectively for Group-4 applications with the DMS schemes. We make two observations. First, both Static-DMS and Dyn-DMS can reduce row energy for Group-4 applications (one outlier is Static-DMS for application `2MM`). Also, Dyn-DMS can more effectively reduce row energy than Static-DMS. Second, both Static-DMS and Dyn-DMS have less than 5% IPC loss. And the IPC of Dyn-DMS can approach closer to 95% of the baseline. We conclude that for applications with low error tolerance, the DMS schemes can still effectively reduce their row energy with no more than 5% IPC loss. Dyn-DMS reduces more row energy by trading off a little more performance.

VI. Related Work

To the best of our knowledge, this is one of the first works in the context of GPUs that consider the interplay between memory scheduling and application's tolerance to latency and errors. Our mechanisms achieve significant memory system energy savings while allowing the underlying hardware to remain dependable both in terms of performance and correctness [32]–[34]. Several prior works in the CPU domain [35]–[42] have focused on improving the row buffer locality. The goal of these works was to reduce the DRAM access latency because it is a first-order performance concern in single-threaded CPU workloads [43]–[45]. Other memory scheduling techniques for CPUs propose to partially delay the write request [39], [40], or conditionally employ an open-row policy [41], [42] to improve the row buffer locality. But the purposes of these works are still to reduce the overall DRAM access latency. In contrast, DRAM access latency is not a primary concern in GPGPU applications as GPUs are capable of hiding long memory access latencies by spawning thousands of concurrent threads. Hence, in this paper, we exploited this property to further enhance the row buffer locality for GPU memory.

In the context of GPUs, Jog et al. [16] proposed a criticality-aware memory scheduling mechanism to trade-off row buffer locality for servicing latency-critical requests. However, it will likely increase the DRAM energy consumption due to sub-optimal row buffer locality. Prior work on warp scheduling and throttling policies [9], [46], [47] can also improve the row buffer locality. However, these throttling/warp-scheduling decisions and memory scheduling decisions do not always remain in sync as they are taken physically far away from each other and are conducted at different granularities. This makes it important to design new memory scheduling decisions (as we do in this paper) that consider the current DRAM status. Moreover, we believe our work is complementary to these prior works as they can provide additional benefits by shaping the access patterns such that they can benefit DMS and AMS.

VII. Conclusions

This paper focused on improving the DRAM row buffer locality in GPUs to reduce the memory system energy consumption. To this end, we proposed a lazy memory scheduler that can work in two modes: delayed or approximate. In the delayed mode, it carefully delays the scheduling of memory requests to allow more of them to accumulate at the memory pending queue. Such a mechanism increases the visibility of the memory scheduler thereby improving the chances of finding more requests that can be served by reusing the data in the row buffer. In the approximate mode, it carefully identifies a small fraction of requests with low row buffer locality and does not issue them to the DRAM banks. Instead, a simple but effective value predictor can be used to approximate the values for such requests. We also find that both these modes are synergistic and improve the effectiveness of each other when employed together. Our evaluation across a variety of GPGPU applications shows that row energy can be reduced by 12% with delayed memory scheduling, 33% with approximate memory scheduling, and 44% with a combination of both schemes. We hope that this paper can open up new research directions that consider the interactions between scheduling, error resilience, and latency tolerance techniques at different levels of the memory hierarchy.

Acknowledgements

We thank our shepherd, Mattan Erez, and anonymous reviewers for their detailed feedback which significantly improved the quality of this paper. We also thank the members of the Insight Computer Architecture Lab for their helpful comments. This material is based upon work supported by the National Science Foundation (NSF) grants (#1657336, #1717532, and #1750667). This work was performed using computing facilities at the College of William & Mary.

REFERENCES

[1] "Top500 Supercomputer Sites - November 2018," *https://www.top500.org/lists/2018/11/*.

[2] "The Green500 List - November 2018," *https://www.top500.org/green500/lists/2018/11/*.

[3] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, 2011.

[4] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation," in *ISCA*, 2014.

[5] J. Trajkovic, A. V. Veidenbaum, and A. Kejariwal, "Improving SDRAM Access Energy Efficiency for Low-power Embedded Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, 2008.

[6] N. Chatterjee, M. OConnor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017.

[7] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems," in *MICRO*, 2017.

[8] D. Wong, N. S. Kim, and M. Annavaram, "Approximating Warps with Intra-warp Operand Value Similarity," in *HPCA*, 2016.

[9] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[10] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger, "Doppelganger: A Cache for Approximate Computing," in *MICRO*, 2015.

[11] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel, "The Bunker Cache for Spatio-Value Approximation," in *MICRO*, 2016.

[12] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "Rfvp: Rollback-free value prediction with safe-to-approximate loads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, 2016.

[13] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[14] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[15] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.

[16] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced Performance in GPUs," in *SIGMETRICS*, 2016.

[17] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.

[18] S. Ghose, A. G. Yaglikçi, R. Gupta, D. Lee, K. Kudrolli, W. X. Liu, H. Hassan, K. K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, and O. Mutlu, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," in *SIGMETRICS*, 2018.

[19] H. Semiconductor, "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0."

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

[21] 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," Sep. 1997.

[22] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.

[23] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management," in *HPCA*, 2018.

[24] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design & Test*, vol. 34, 2017.

[25] L.-N. Pouchet, "Polybench: the Polyhedral Benchmark Suite," *URL: http://www.cs.ucla.edu/~pouchet/software/polybench/*, 2012.

[26] J. S. Miguel, M. Badr, and E. N. Jerger, "Load Value Approximation," in *MICRO*, 2014.

[27] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware," in *OOPSLA*, 2013.

[28] D. Mahajan, K. Ramkrishnan, R. Jariwala, A. Yazdanbakhsh, J. Park, B. Thwaites, A. Nagendrakumar, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Abstractions for Approximate Hardware Design and Reuse," *IEEE Micro*, vol. 35, 2015.

[29] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *PLDI*, 2011.

[30] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi *et al.*, "Axilog: Language Support for Approximate Hardware Design," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[31] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "Flexjava: Language Support for Safe and Modular Approximate Programming," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[32] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A Large-scale Study of Soft-errors on GPUs in the Field," in *HPCA*, 2016.

[33] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation," in *HPCA*, 2015.

[34] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications," in *MICRO*, 2018.

[35] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality-aware Data Placement in Hybrid Memories," in *ICCD*, 2011.

[36] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality," in *MICRO*, 2000.

[37] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing DRAM Efficiency with Locality-aware Data Placement," in *ASPLOS*, 2010.

[38] Z. Zhang, Z. Zhu, and X. Zhang, "Breaking Address Mapping Symmetry at Multi-levels of Memory Hierarchy to Reduce DRAM Row-buffer Conflicts," *The Journal of Instruction-Level Parallelism*, vol. 3, 2001.

[39] Y.-S. Moon, Y. Kwon, H.-S. Kim, D.-g. Kim, H. H. Lee, and K. Park, "The Compact Memory Scheduling Maximizing Row Buffer Locality," in *3rd JILP Workshop on Computer Architecture Competitions: Memory Scheduling Championship*, 2012.

[40] C. Natarajan, B. Christenson, and F. Briggs, "A Study of Performance Impact of Memory Controller Features in Multi-processor Server Environment," in *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, 2004.

[41] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "HAPPY: Hybrid Address-based Page Policy in DRAMs," in *Proceedings of the Second International Symposium on Memory Systems*, 2016.

[42] "Introducing Intel's Adaptive Page Management Technology," *https://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask/6*.

[43] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.

[44] V. Cuppu and B. Jacob, "Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-system Performance?" in *ISCA*, 2001.

[45] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "ANATOMY: An Analytical Model of Memory System Performance," in *SIGMETRICS*, 2014.

[46] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[47] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.