Prefetching Techniques for Near-memory Throughput Processors

Reena Panda*
University of Texas at Austin
reena.panda@utexas.edu

Yasuko Eckert
AMD Research
yasuko.eckert@amd.com

Nuwan Jayasena
AMD Research
nuwan.jayasena@amd.com

Onur Kayiran
AMD Research
onur.kayiran@amd.com

Michael Boyer
AMD Research
michael.boyer@amd.com

Lizy Kurian John
University of Texas at Austin
ljohn@ece.utexas.edu

ABSTRACT

Near-memory processing or processing-in-memory (PIM) is regaining a lot of interest recently as a viable solution to overcome the challenges imposed by memory wall. This trend has been mainly fueled by the emergence of 3D-stacked memories. GPUs are touted as great candidates for in-memory processors due to their superior bandwidth utilization capabilities. Although putting a GPU core beneath memory exposes it to unprecedented memory bandwidth, in this paper, we demonstrate that significant opportunities still exist to improve the performance of the simpler, in-memory GPU processors (GPU-PIM) by improving their memory performance. Thus, we propose three light-weight, practical memory-side prefetchers to improve the performance of GPU-PIM systems. The proposed prefetchers exploit the patterns in individual memory accesses and synergy in the wavefront-localized memory streams, combined with a better understanding of the memory-system state, to prefetch from DRAM row buffers into on-chip prefetch buffers, thereby achieving over 75% prefetcher accuracy and 40% improvement in row buffer locality. In order to maximize utilization of prefetched data and minimize thrashing, the prefetchers also use a novel prefetch buffer management policy based on a unique dead-row prediction mechanism together with an eviction-based prefetch-trigger policy to control their aggressiveness. The proposed prefetchers improve performance by over 60% (max) and 9% on average as compared to the baseline, while achieving over 33% of the performance benefits of perfect-L2 using less than 5.6KB of additional hardware. The proposed prefetchers also outperform the state-of-the-art memory-side prefetcher, OWL by more than 20%.

*Author contributed to this work during her internship with AMD Research.

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926282

CCS Concepts

•**Computer systems organization** → *Single instruction, multiple data;*

Keywords

Processing-in-memory; GPU; Prefetching; 3D die-stacked memory

1. INTRODUCTION

Processing-in-memory (PIM) or near-memory processing has been a topic of great interest among researchers since the mid-90s [42, 19, 41, 12, 22], touted as solutions that can bridge the performance gaps between processing and memory speeds (*"memory wall"*) by bringing computation closer to where data resides. However, challenges involved in integrating memory and logic had prevented them from becoming the mainstream processing paradigm. More recently, PIM systems are regaining a lot of interest [50, 8, 7, 11, 47], a trend that is mainly fueled by the emergence of 3D stacked memories [4, 5, 6], which enable the cost-effective integration of logic processors and memory. Thermal challenges, however, impede stacking memory on top of high-performance processors. Therefore, several recent research studies [50, 11] have proposed PIM systems incorporating simpler processors in the logic layer of the memory stacks. Figure 1 shows such a system organization where in-memory GPGPU processors (GPU-PIM) are stacked under the memory dies to better leverage the increased bandwidth provided by stacked DRAMs [15, 50].

GPGPUs are touted as great candidates for in-memory logic processors due to their superior bandwidth utilization capability, energy-efficiency and programmability fea-

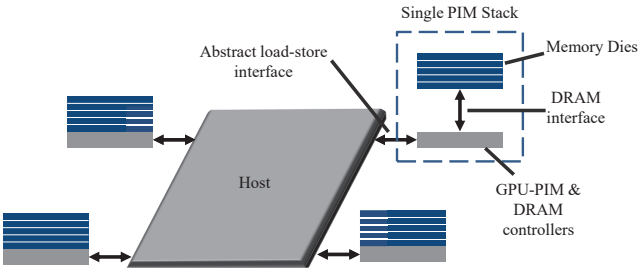

Figure 1: An example processing-in-memory system

tures. But can in-memory GPGPU processors exploit the immense memory-level parallelism opportunities provided by die-stacking to completely overcome the challenges imposed by the memory wall? To study the impact of memory performance on the performance of a GPU-PIM stack, we conducted an experiment where we made the GPU's last-level caches perfect. Figure 2 shows the performance improvements across a suite of GPGPU applications. We can observe that for a wide variety of applications like SRAD, streamcluster and Triad, speedups as high as 1.7x can be obtained by improving the memory performance of the in-memory processors. Thus, the perfect-memory performance still shows significant scope to improve performance across a vast majority of applications.

Putting a core beneath memory exposes unprecedented memory bandwidth to the core, and we argue that the inherent mismatch between potential memory-level parallelism and available compute-level parallelism using the simpler in-memory processors significantly limits their performance gains. GPGPUs employ fine-grained multi-threading and on-chip caching to hide the impact of long-latency memory operations. However, increasing the thread-level parallelism in the in-memory processors would also result in significant increase in power (higher thermal concerns) due to increased resource requirements (larger register files, state for wavefront-scheduling and divergence, etc.) to support higher degree of multi-threading and may also affect performance due to increased contention in the on-chip resources. In this paper, we propose light-weight prefetching schemes to improve the memory-latency hiding capability and performance of in-memory GPGPU cores in an energy-efficient manner. We specifically focus on memory-side prefetching techniques, which fetch data from the memory system but do not push the data into the caches/cores before receiving demand requests.

GPGPU applications exhibit significant synergy and regularity in the memory accesses made by threads both within and across wavefronts. The proposed heuristics not only exploit patterns in the past history of individual memory accesses but also the intra- and inter-wavefront access locality/predictability to improve prefetching accuracy. Unlike core-side prefetching schemes, the proposed memory-side prefetching heuristics also leverage the prefetcher's proximity to the memory system and a better understanding of the current memory system state to prefetch data from DRAM row buffers into on-chip prefetch buffers, thereby significantly improving row buffer locality. Choosing the right prefetch buffer management and prefetch-timing policy is also equally critical in order to ensure optimal performance gains. Using a traditional LRU-based buffer management policy would not work because several concurrent memory access streams would cause significant thrashing in the prefetch buffer. In order to maximize utilization of prefetched data, we propose a novel "dead-row prediction" mechanism to predict the degree of "liveness" of a currently cached row, which is then used to control eviction from the prefetch buffer. Further, to minimize thrashing effects, we use an eviction-based prefetch trigger (i.e., we issue prefetch requests only if there is space in the prefetch buffer).

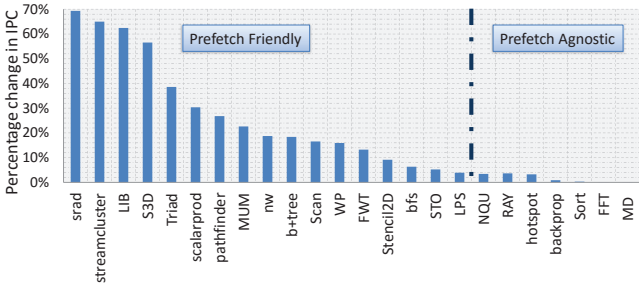

Figure 2: IPC Improvement with Perfect-L2 on GPU-PIM stack

Proposed prefetcher-enabled designs improve the latency-tolerance capability of in-memory GPGPU processors and thus, they can serve a diverse set of applications while better leveraging the bandwidth provided by 3D-stacked systems. The key contributions made in this paper are:

- We propose light-weight, memory-side prefetching techniques to improve the performance and energy efficiency of GPU-based PIM systems that leverage their knowledge of the current state of the memory system to prefetch data from row buffers, thereby improving row buffer locality by over 40%.
- We propose a novel prefetch buffer management mechanism and prefetch-timing policy based on a unique dead-row prediction mechanism that maximizes utilization of prefetched data and minimizes thrashing.
- We also exploit the synergy in the wavefront-localized memory access streams to improve the accuracy and timeliness of the prefetchers.
- Our experimental results demonstrate that proposed prefetchers improve performance by over 60% (max) and 9.3% on average over the baseline, achieving over 33% of the performance benefits of perfect-L2 with only 5.6KB of additional hardware overhead. The proposed prefetchers also outperform the state-of-the-art memory-side prefetcher, OWL by more than 20%. We also demonstrate the applicability of the prefetchers to other GPGPU configurations.

The rest of this paper is organized as follows: In Section 2, we provide a brief background about our baseline system and GPU prefetching. In section 3, we describe the proposed prefetching heuristics followed by discussing our experimental framework and evaluation in sections 4 and 5 respectively. Section 6 discusses prior work and section 7 concludes the paper. Without loss of generality, we will refer to GPGPU as GPU in the rest of this paper.

2. BACKGROUND

This section describes the baseline GPU architecture and provides a brief background about prefetching in GPUs.

2.1 Baseline Architecture

GPUs consist of multiple SIMD cores (streaming multiprocessors (SMs) in NVIDIA GPUs and compute units (CU) in AMD GPUs). Each CU fetches, decodes a group of threads (warps in NVIDIA GPUs or wavefronts in AMD GPUs) and executes them in lockstep, following a single instruction multiple thread (SIMT) model. Each CU is associated with a private L1 data-cache, read-only texture cache,

constant caches and shared memory. The global memory is partitioned and all CUs are connected to the partitioned modules by an interconnection network. Each memory controller consists of a slice of the shared L2 cache and the DRAM partition. We assume write-back policies for both L1 and L2 caches and a minimum L2 miss latency of 120 compute core cycles [26, 48].

2.2 Baseline GPU-PIM System

Our target GPU-PIM architecture stacks a GPU processor under high bandwidth memory (HBM). The GPU processor is designed based on area considerations of a typical 3D-memory stack. Assuming the die-size of a 8Gb HBM2 die is in the range of 40-50mm^2 [36], we modeled our baseline GPU-PIM architecture to be a scaled-down version of the NVIDIA Fermi GPU [2] and estimated the CU count and L2 size of the GPU logic layer after appropriate scaling to 20nm technology [1]. The target system is shown in Figure 5 (described later).

2.3 Inefficiencies in GPU Cache Performance

GPU execution model leverages thread-level parallelism and on-chip caching to hide the impact of long memory-access latencies. The high degree of thread concurrency, however, leads to significantly lower effective cache-space per thread which causes significant contention in the on-chip caches. Figure 3 shows the last-level cache (LLC) miss rates of the prefetch-friendly applications. We can see that most applications suffer from high LLC miss rates, over 69% on average. Prior research studies [34, 35, 39] have also reported similar observations and thus, GPU LLCs tend to function as memory bandwidth filters. The perfect-L2 performance speedup results (Figure 2) discussed before also demonstrate that there exists significant opportunity to improve GPU performance by improving memory performance. In this paper, we explore prefetching to do the same. Every opportunity comes with challenges and GPU prefetching comes with its own set of challenges. The following section describes some of the key challenges in designing prefetching solutions for GPUs.

2.4 Challenges with GPU Prefetching

Although GPU applications are traditionally considered to have highly regular memory access behavior, designing prefetching schemes for GPUs is tricky because naively adding prefetching support can hurt their performance [32, 49]. Also, many recently proposed GPU applications [18, 16] exhibit irregular memory access behavior and thus, naive prefetching schemes cannot be applied for such applications.

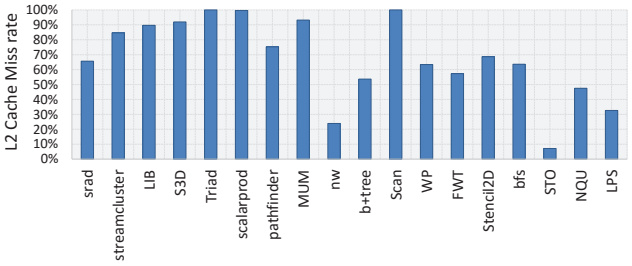

Figure 3: L2 miss rates across prefetch-friendly benchmarks

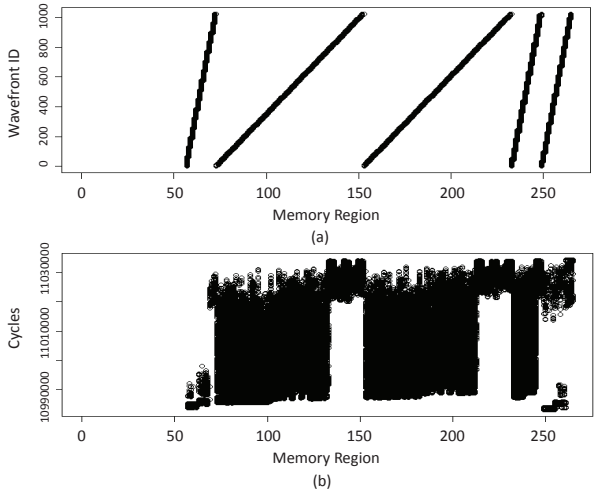

Figure 4: Memory access patterns with respect to (a) wavefronts (b) cycles for SPMV application

As a result, hardware prefetching techniques are typically not employed in mainstream GPUs [9, 34].

Figure 4 shows the memory access patterns for sparse matrix vector multiplication (SPMV) application. The x-axis shows the memory regions (defined as contiguous 16KB segments in the physical address space) accessed by the application. Figure 4a shows the wavefronts (y-axis) that access the corresponding memory regions and Figure 4b shows the execution cycles (y-axis) when the memory regions are accessed. We can see that even though there is significant spatial locality in the memory access streams localized across the wavefronts, multiple concurrently-active memory streams cause significant interleaving in the access patterns which complicates runtime pattern detection. Furthermore, it leads to increased contention in the on-chip caches and increases the likelihood of premature eviction of prefetched data. Although die-stacking provides significant memory bandwidth, aggressive prefetching can hurt performance by conflicting with demand requests and creating contention in the already loaded buses and memory system. Thus, designing a memory-side prefetcher for GPUs is challenging.

2.5 Overcoming the Challenges

Several prior studies have explored core-side prefetching [32, 44, 25] and memory-side prefetching [26] in GPUs. To overcome the challenges discussed before, prior studies have proposed to throttle prefetcher aggressiveness based on its accuracy [32], adopted stringent constraints to control the accuracy of the prefetcher [44] or throttled wavefront scheduling policies to limit the degree of multi-threading thereby, reducing contention in on-chip caches [28, 26]. We will discuss these proposals in detail in Section 6.

In this paper, we propose memory-side prefetching schemes to improve the performance of GPU-PIM systems. To enable accurate pattern detection in spite of the highly interleaved memory accesses, the prefetchers analyze memory accesses at a higher granularity and leverage their proximity to the memory system to prefetch from DRAM row buffers, thereby improving row buffer locality. As discussed, GPU applications exhibit significant regularity in the wavefront-localized memory access streams. The heuristics exploit not

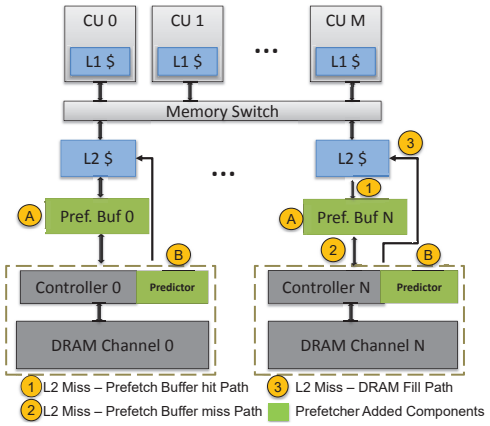

Figure 5: Prefetcher-enabled architecture overview

only the past history of individual memory accesses but also the intra- and inter-wavefront patterns to improve prefetcher accuracy. In order to minimize contention, we prefetch into a separate prefetch buffer, which is managed using a novel management policy that maximizes utilization of prefetched data. Finally, our prefetch-timing policy uses the prefetch buffer occupancy and usefulness information as a trigger to initiate future prefetches to control prefetcher aggressiveness.

3. PREFETCHER DESIGN

Figure 5 shows the overall architecture of the GPU-PIM stack equipped with memory-side prefetcher. As shown in the figure, we prefetch into a separate prefetch buffer Ⓐ that sits in the L2-cache miss-path ①. The prefetcher/predictor Ⓑ is implemented in the memory controller and leverages its proximity to the memory system to monitor the memory access streams and make prefetch decisions. L2 misses that hit in the prefetch buffer are satisfied from the prefetch buffer itself. Prefetch buffer read misses and write accesses are forwarded to the DRAM system. There is a separate fill path for servicing prefetch requests ② and demand requests ③.

There are three primary design considerations for implementing a data prefetcher: (DC-1) choosing the prefetch granularity (e.g., single cacheline, next-N lines, etc.), (DC-2) identifying what to prefetch and (DC-3) deciding when to initiate the prefetch and the prefetched-data management policy. In this paper, we propose three prefetching heuristics to address these design considerations leveraging key insights derived from extensive workload characterization. The first proposal, **"locality-aware prefetcher"**, leverages the low-latency and energy-efficient access of DRAM row buffers (DC-1) coupled with the application's past access history (DC-2) and a novel prefetch-timing and buffer management policy (DC-3) in order to make its prefetching decisions. The second proposal, **"wavefront-correlation-aware prefetcher"**, builds upon the first heuristic and improves the prefetcher accuracy (DC-2) by exploiting the synergy and regularity in the wavefront-localized memory access streams. Finally, **"cacheline-reuse-aware prefetcher"** is proposed as an enhancement over the previous two schemes that exploits the temporal re-usability characteristics of applications to manage the prefetch buffer space more efficiently (DC-3).

3.1 Locality-aware Prefetcher Design

This section describes the locality-aware prefetcher in detail.

3.1.1 Picking the prefetch granularity

Observation: As discussed, GPU applications suffer from high LLC miss rates, which also leads to more concurrent memory requests. DRAM-based devices employ peripheral storage structures called row buffers (one per DRAM bank), which provide lower-latency and energy-efficient access to data as compared to accessing the memory array. Higher number of concurrent requests from multiple memory streams lead to poor row buffer locality. We observed that for several applications such as mummerGPU (MUM), LIB, scalarProd, etc., the row buffer conflict rate (percentage of row buffer misses over DRAM accesses) gets as high as 60 - 80%, which causes significant performance degradation in the memory system.

To design techniques that improve row buffer locality, we studied spatial locality in DRAM rows across several GPU applications. A DRAM row, in our target organization, consists of 32 cachelines. Figure 6 shows the percentage of rows accessed during the benchmark execution that have a certain *"L"* number of unique cachelines touched. For most applications, significant spatial locality exists within DRAM rows, which can be exploited to prefetch at row granularity while leveraging the low-latency and energy-efficient access of DRAM row buffers. Prefetching at row granularity also enables coarse-grained tracking with less area overhead to track the same number of entries as compared to cacheline granularity.

Proposed design: Locality-aware prefetcher exploits the lower latency and energy-efficient access of row buffers to prefetch *candidate* DRAM rows, thereby improving overall performance and energy-efficiency of applications. Although prefetching entire rows causes higher memory-bandwidth consumption, the use of in-package stacked DRAMs combined with intelligent prefetching policies allows us to do so efficiently. Prefetch candidate rows are identified by learning spatial and temporal locality properties in the memory access streams, as we will discuss later. As we prefetch at a higher granularity, it is important to utilize the prefetched data, otherwise performance may degrade due to high number of prefetches. This makes it important to devise appropriate prefetch-timing policy and prefetched data management policy, which we will discuss next.

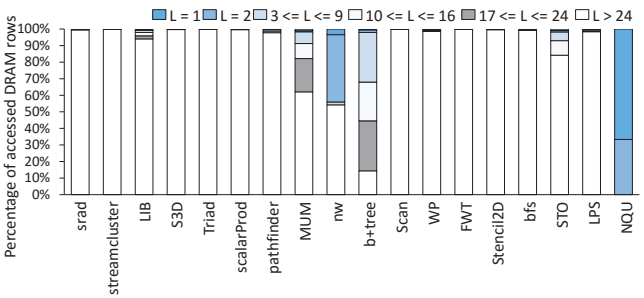

Figure 6: Spatial locality: Percentage of rows with L lines touched

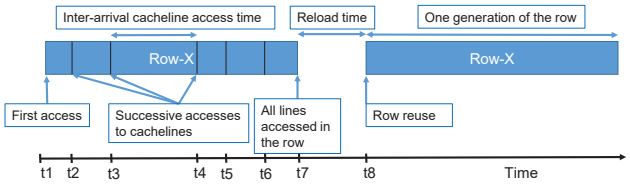

Figure 7: Row access lifetime characterization

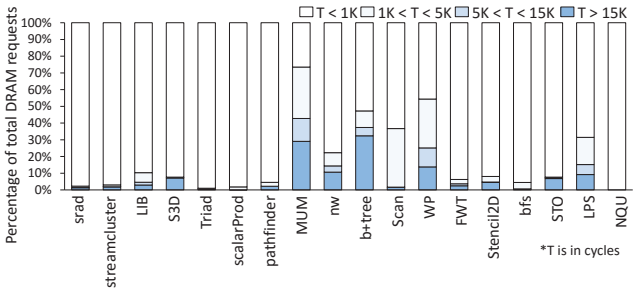

Figure 8: Inter-arrival cacheline access time characterization

3.1.2 Deciding the prefetch buffer management policy

Given the limited size of the prefetch buffer (∼4-5 rows), designing an appropriate prefetch buffer management and prefetch-timing policy is important to optimize performance gains. If data is not prefetched in a timely manner or if prefetched data gets evicted pre-maturely, it can lead to sub-optimal performance or can even be detrimental to performance due to increased contention in the memory system, buses, etc. Ideally, if a *high spatial-locality* row is prefetched into the prefetch buffer, it should get evicted only when it is not expected to receive any more demand requests within a reasonable time (i.e., after the row is practically dead). Given the limited size of the prefetch buffer, using a traditional LRU-based buffer management policy would not work because the multiple, concurrently-active memory access streams would cause thrashing in the prefetch buffer and hurt performance. Thus, in order to maximize utilization of prefetched data, we propose a novel ***"dead-row prediction"*** scheme to predict the liveness of a cached row, which is used to explicitly control the prefetch buffer eviction policy. To minimize thrashing effects, we also use an ***"eviction-based prefetch trigger"*** (i.e., we issue prefetches depending upon prefetch buffer space availability).

The dead-row prediction scheme is designed by analyzing the row access patterns and timing characteristics across a suite of GPU applications. The underlying idea can be explained using the row access timeline shown in Figure 7 (for row X): Row X is accessed first at time t1. It receives successive demand requests (to same or different cachelines) at times t2, t3, etc. At time t7, all the unique cachelines in row X have each received at least one demand request. Earlier, we saw that over 95% of the accessed rows in most applications have high spatial locality and thus, they receive at least one demand request for every unique cacheline. We define the time-interval between the first access to a row to an access to the last unique cacheline within the row as one generation of the row. Thereafter, the row may get reused [i.e., receive more demand requests (shown at time t8)] or not depending upon the application's characteristics. The dead-row prediction mechanism builds upon this row access lifetime characterization and leverages (and defines) two key timing metrics, namely the *inter-arrival cacheline access time* (or inter-arrival time) and *reload time*, to predict the degree of liveness of a currently cached row as we will describe next. Similar schemes have been explored in prior studies [29, 30, 37, 24] but in the context of individual cachelines/blocks in CPU caches.

Inter-arrival cacheline access time: It is defined as the time interval between successive accesses to a row within a single generation. An estimate of the average inter-arrival cacheline access time (say *CaTh* cycles) can be used to identify dead rows depending upon if the row has received any demands in the last *CaTh* cycles. Figure 8 shows the inter-arrival cacheline access time for several GPU applications. We can see that most benchmarks have ∼80% of their requests arrive with an inter-arrival time of less than 1K cycles. Few others such as b+tree, MummerGPU, WP have higher inter-arrival access times (mostly less than 15K cycles).

Reload time: It is defined as the time-interval between successive generations of a row. It can be used to predict the usefulness in caching a row even after all its unique lines are accessed depending upon if the row is going to get reused soon. Figure 9 shows the reload time for several GPU applications. Applications mostly fall into three categories:- (a) Applications with most rows having short reload time (e.g., srad, bfs, LIB, etc.), (b) Applications with most rows having very high reload time (e.g., FWT, streamcluster) and (c) Applications where rows are mostly not reused (e.g., b+tree, pathfinder, scalarProd). For category (b) and (c) applications, it is optimal to replace a row immediately after a generation completes as the row would either be reused after a significantly long time or never be reused at all. For applications in category (a), it is better to continue caching the row even after a generation as it would mostly be reused shortly.

Dead-row predictor implementation: The inter-arrival time and reload time thresholds are used for implementing dead-row prediction. At any time, only a fixed number of rows (MAX_ROW) are prefetched to avoid contention and thrashing effects. After a cached row is identified to be dead, it is evicted from the prefetch buffer making room for a new candidate row (eviction-based prefetch trigger). In our experiments, we use a reload time threshold of 256 cycles. For the inter-arrival time, we support two thresholds, a lower threshold of 1K cycles and a higher threshold of 15K cycles

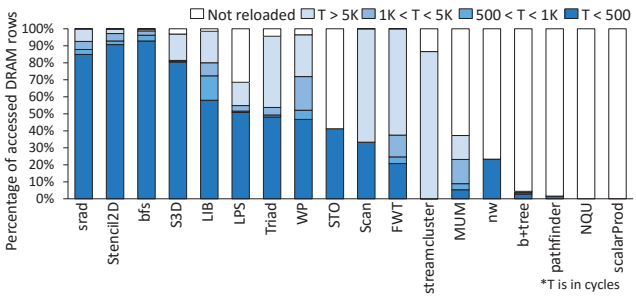

Figure 9: Reload time characterization

Row Tag	Valid	Line BV	Demands	Conflicts	Ref-Ctr	Pfed
14-bits	1-bit	32-bits	6-bits	6-bits	6-bits	1-bit

Figure 10: Row tracking table entry

and we choose the appropriate threshold for an application based on training on the access patterns for the first 10K requests. Later, we will discuss the prefetcher sensitivity to different threshold values.

Small amount of hardware can track the time-keeping metrics. One coarse-grained counter (ref-ctr) per tracked row, which is ticked periodically with a global cycle counter, can be used to measure the reload/inter-arrival time. The counter is reset on every access to the row and is advanced by 1 with a global tick that occurs every 256 cycles. The counter value roughly represents the time elapsed since the last access to the row. If all the lines in a row are touched (by ANDing the cacheline bit-vector), then a counter value of more than 1 (reload-time threshold) implies that a generation has ended and the row's reload time is not short. If all lines are not touched, a counter value of 4 or more can be used to imply that the row is dead assuming an inter-arrival time threshold of 1K cycles.

3.1.3 Identifying the prefetch candidates

In order to identify prefetch-candidate rows, locality-aware prefetcher leverages on the observation that most GPU applications have high spatial locality at DRAM row granularity. We also observed that rows that received more demand requests (or/and caused more row buffer conflicts) within a given period of time have a higher chance of receiving more demand requests (or/and causing more row buffer conflicts) in near future. To capitalize on these observations, the locality-aware prefetcher monitors accesses to a set of recently touched rows in the row tracking table (RTT). Each RTT entry (in Figure 10) monitors the row's unique cacheline accesses using a 32-bit vector (*line BV*), the row's *hotness* using the number of demands to the row, the number of row buffer conflicts caused by accesses to this row and the row's access timing information (*ref-ctr*). The prefetching heuristic picks the next-to-prefetch row from a pool of candidate rows using the following policies:

- Prioritize rows with the least number of unique cachelines touched
- Prioritize rows with highest "Row Weight"

$$RowWeight = Conflicts * \alpha + Hits$$

The first condition prioritizes rows that have the least number of cachelines touched so far and thus, have a higher chance of receiving demand requests for the untouched cachelines in future. The second condition (α is empirically determined to be 3 for our configuration) gives higher weightage to performance-critical, high-demand (*Hits* is defined as *Demands* - *Conflicts*) rows that have caused more row buffer conflicts in the past. In order to improve row buffer locality, the locality-aware prefetcher first attempts to pick the next-to-prefetch row from candidate rows (if tracked in RTT) that have demand requests already queued in the memory controller using above prefetching policies. In case there are no outstanding memory requests, the predictor picks the next-to-prefetch row from all the rows tracked in the RTT.

3.1.4 Prefetcher implementation

Locality-aware prefetcher enables selective caching of a set of performance-critical DRAM rows thereby maximizing the utilization of prefetched data. The predictor employs the RTT to track the access locality of a few recently accessed rows. On a prefetch buffer miss, the RTT entry is accessed and its corresponding fields are updated if the row is tracked. If not, a new row is allocated in the RTT to be tracked depending upon space availability. If a tracked row is identified to be dead, then the corresponding RTT entry is freed. It is to be noted that RTT tracking is not on the critical path of servicing a prefetch buffer miss request as the RTT meta-data is required only when a new prefetch candidate has to be identified (occurs after >1K cycles). If either less than MAX_ROW number of rows are cached in the prefetch buffer or a currently cached row is identified to be dead at any time, a new row is prefetched using the heuristic described before. The prefetch row identification heuristic is a simple but effective technique that exploits past row access history to identify future prefetch candidates. However, it suffers from a limitation that it cannot predict first access to rows which our next prefetcher proposal tries to overcome.

3.2 Wavefront-correlation-aware Prefetcher Design

The wavefront-correlation-aware prefetcher builds upon the locality-aware prefetcher and tries to overcome its limitation of not being able to predict first access to rows. In order to improve prefetch timeliness by predicting first row accesses, the wavefront-correlation-aware prefetcher exploits the regularity in the memory access patterns localized at individual wavefront granularity. Earlier, we had showed in Figure 4a that significant spatial locality and thus, better predictability exists in the memory access streams accessed by different wavefronts. We specifically leverage two kinds of patterns: intra-wavefront (intra-wf) locality (defined as the locality in the row access patterns within threads of

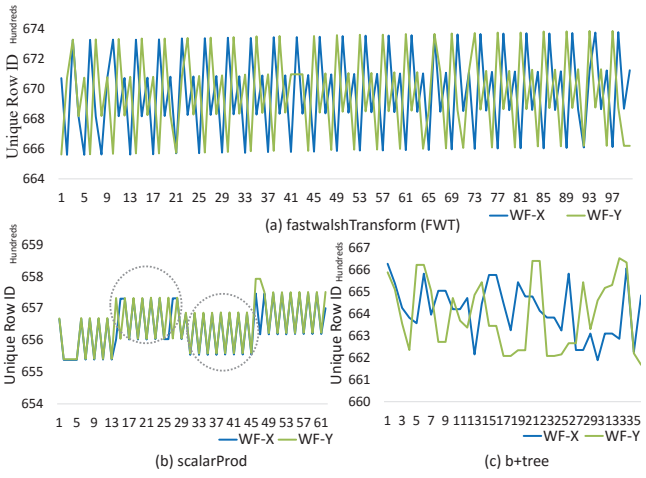

Figure 11: Unique row access patterns for two wavefronts in benchmarks: (a) FWT (b) scalarProd and (c) b+tree

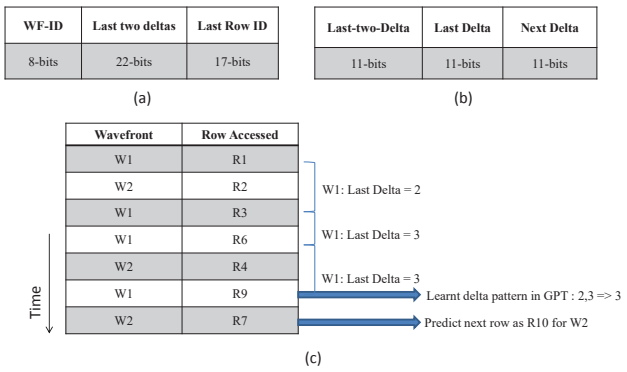

Figure 12: Figure showing (a) WF tracking table (b) Global pattern table (c) Example delta-pattern

a single wavefront) and inter-wavefront (inter-wf) locality (refers to the similarity and repeatability in the row access streams across different wavefronts). This prefetcher proposes a new prefetch candidate selection logic, the other two design choices (DC-1 and DC-3) stay the same as the previous scheme.

3.2.1 Observation

Figure 11 shows the row accesses made by two wavefronts for FWT, scalarProd and b+tree applications. Successive accesses to the same row by a wavefront is shown as a single access. The x-axis shows the sequence of row accesses made by the two wavefronts, while y-axis shows the unique row-ids. FWT shows good predictability in its row accesses, both within (intra-wf locality) and across (inter-wf locality) the two wavefronts. ScalarProd benchmark has good intra-wf locality within individual phases (marked by dotted circles) but not so much across phases. Nevertheless, it still demonstrates good inter-wf locality as shown by its access characteristics for the two wavefronts. On the contrary, b+tree, an irregular application, exhibits neither good intra- nor inter-wf locality. The wavefront-correlation-aware prefetcher leverages the spatial locality and predictability in the wavefront-localized row access patterns for benchmarks that have good intra- and inter-wf locality properties in order to improve its prefetch accuracy and timeliness.

3.2.2 Prefetcher design

The wavefront-correlation-aware prefetcher introduces two additional structures, (a) wavefront tracking table (WFT) and (b) global pattern table (GPT) as shown in Figure 12a and 12b respectively. Instead of tracking patterns across a large number of wavefronts which would lead to larger number of prefetches and thus, higher contention, WFT monitors memory accesses made by a few wavefronts, which do not access the same rows exactly to maximize coverage. The prefetcher continuously learns delta patterns in the memory access streams of the tracked wavefronts, where delta is defined as the difference between successive unique row accesses made by a wavefront. WFT specifically learns two-history delta correlation in the row access streams as we observed that single-history delta correlation had lower accuracy due to shorter history depth. An example of the learning process is shown in Figure 12c. When a tracked wavefront learns a new delta pattern, it updates the same

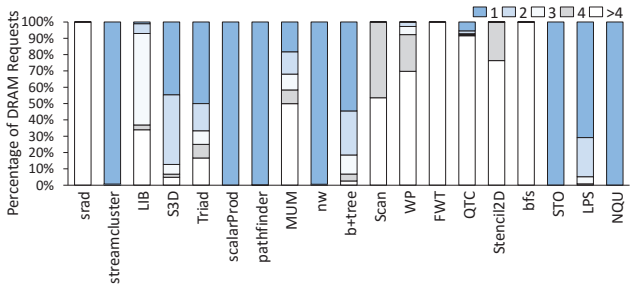

Figure 13: Cacheline reuse characterization

in the GPT. GPT is a delta-correlating table, which tracks the two-history delta patterns. GPT is shared across all wavefronts, and helps to exploit inter-wf locality by learning patterns on a few wavefronts and then using them to make prefetch predictions for other wavefronts. The wavefront-correlation-aware prefetcher exploits delta-correlation in row access streams where strong patterns exist, and controls the prefetcher aggressiveness for applications (e.g., b+tree) with weak delta correlations. This prefetcher works well if strong delta correlation exists. If not, this technique might cause useless row prefetches.

3.3 Reuse-aware Prefetcher Design

To maximize utilization of prefetched data and minimize memory bandwidth waste, the locality-aware and wavefront-correlation-aware prefetchers cache a row in the prefetch buffer till the row is predicted to be *dead* and employ an eviction-based trigger to initiate new row prefetching. However, in case there is limited re-usability of individual cachelines within a cached row, higher performance gains can be achieved by exploiting this characteristic to manage the prefetch-trigger/timing policy more effectively. The reuse-aware prefetcher is proposed as an enhancement over the prior two heuristics that leverages the cacheline-level reusability and temporal locality characteristics to enable better utilization of the prefetch buffer space.

3.3.1 Observation

Different applications have different temporal locality properties at individual cacheline granularity. Many benchmarks access particular memory regions just once, while many others access memory regions multiple times but the reuse distance is short enough that successive requests to the data

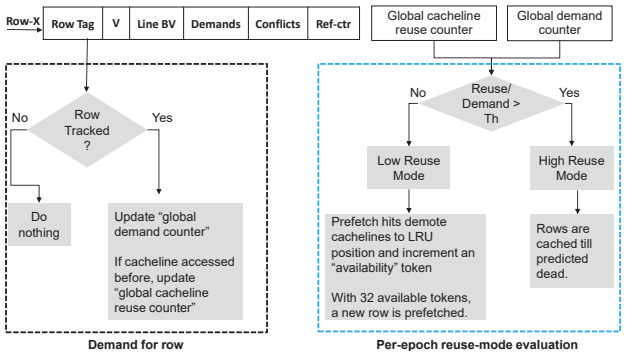

Figure 14: Reuse-aware prefetcher algorithm

cachelines get serviced in the cache hierarchy. Figure 13 shows the demand requests (L2 misses) received by individual cachelines for several applications. For applications such as scalarProd, streamcluster, pathfinder, etc. that access cachelines mostly once, keeping the entire row cached in the prefetch buffer until it is completely dead leads to sub-optimal performance gains.

3.3.2 Proposed design

To detect and leverage low temporal locality of certain applications, the reuse-aware prefetcher allows two modes of operation: (a) High cacheline reuse mode where individual cachelines are reused multiple times, (b) Low cacheline reuse mode where most cachelines are not reused. The mode of operation is chosen on a per-epoch basis (where an epoch is defined as 10K memory accesses) using two global counters per memory controller, one counter to track global demands to tracked rows and the other counter to track cacheline reuse in the tracked rows. Reuse can be identified by monitoring the cacheline bit-vector tracked per RTT entry. The reuse-aware algorithm is shown in Figure 14. Operation starts in high-reuse mode. The predictor monitors the global demand and reuse counters during every epoch. At the end of each epoch, the fraction of reuse over demands is computed (a costly divide operation but done once every thousands of cycles) and compared against a reuse threshold (0.3 in our experiments). If the value is below the reuse threshold, then operation switches to the low reuse mode. The reuse-aware prefetcher manages the prefetch buffer at cacheline granularity (16-way set-associative). In the low reuse mode, as cachelines receive demand requests in the prefetch buffer, they get demoted to the low-priority position and an *availability-token* is incremented. Once 32 tokens (number of cachelines in a row) become available, a new row prefetch is triggered. In order to limit the maximum number of rows that can be prefetched simultaneously, we limit the maximum number of rows that can be resident in the prefetch buffer simultaneously to twice the MAX_ROW size. This proposal allows better use of available prefetch buffer space and can be applied to both locality-aware and wavefront-correlation-aware prefetcher designs. However, it increases the hardware complexity (tag-space) of the prefetch buffer which needs to track data at individual cacheline granularity because cachelines within a row are no longer kept contiguously in the prefetch buffer.

4. EXPERIMENTAL EVALUATION

Simulation infrastructure: The proposed prefetcher designs are evaluated using GPGPUsim V3.2.2 [10], which is a widely used cycle-level simulator for GPU architecture research. Figure 5 shows the overall architecture of the GPU-PIM stack (combining HBM memory with a GPU processor in its logic layer) with a memory-side prefetcher. The baseline PIM architecture is modeled based on a scaled-down version of the NVIDIA Fermi GPU [2] accounting for area considerations of the HBM stack. We assume the die-size of an 8Gb HBM2 die to be in the range of 40-50mm^2 [36]. Based on this size estimate, we estimated the CU count and L2 size of the GPU-logic layer and the configuration is shown in Table 1.

Table 1: Baseline Configuration

Compute Units	6 SMs, 1400MHz
L1 Cache/Shared Memory	16 KB/48 KB, 128 B line, 1-cycle/3-cycle hit latency
L2 Cache	128 KB, 8 banks, 128 B line, 8-way associativity
DRAM	8 Memory Channels, 1 Rank/Channel, 8 Banks/Rank, BW – 256 GB/sec, 924 MHz, tRCD-tCAS-tRP-tRAS: 11-11-11-28
DRAM schedule Queue	Size = 16 and out-of-order (FR-FCFS scheduling policy)
Wavefront scheduling policy	Greedy-then-oldest (GTO)
Prefetch Buffer	16KB/Memory Channel, 2-cycle hit latency

Table 2: Benchmarks

Rodinia	Srad, streamcluster, pathfinder, needleman-wunsch(nw), b+tree, bfs, hotspot*, backpropagation*
ISPASS-2009 Suite	LIB, MummerGPU, Weather Prediction (WP), StoreGPU (STO), Laplace (LPS), N-Queens (NQU*), Ray Tracing (RAY*)
SHOC Suite	S3D, Triad, Scan, Stencil2D, Sort*, FFT*, MD*
gpuComputingSDK	scalarProd, fastWalshTransform (FWT), blackshcoles*

*Prefetch Agnostic Benchmarks

Benchmarks: In order to represent a wide variety of optimized real-world GPU applications, we study 25 applications from several popular GPGPU benchmarking suites like Rodinia [17], NVIDIA SDK [3], SHOC [18] and GPGPU-sim ispass-2009 [10] benchmark suites as shown in Table 2. We execute these applications in GPGPU-Sim, which simulates the baseline GPU-PIM architecture described in Table 1. The applications are run until completion or for 1 billion instructions whichever comes first.

5. RESULTS AND ANALYSIS

In this section, we evaluate and compare the effectiveness of the proposed prefetching techniques.

5.1 Instructions Per Cycle

Figure 15 shows the IPC improvements obtained as compared to a baseline no-prefetcher system across the prefetch-friendly benchmarks, for a Perfect-L2 system, a system with twice the size of L2 cache as the baseline organization (2X-L2), the state-of-the-art memory-side prefetcher, OWL [26] with twice the size of the baseline L2-cache and the three proposed prefetching techniques: the locality-aware prefetcher (Loc-PF), the wavefront-correlation-aware prefetcher (LocWF-PF) and the cacheline-reuse-aware enhancement over the LocWF-PF technique (LocWFCL-PF).

The 2X-L2 configuration improves performance over the baseline configuration by less than 2% on average showing that intelligent-schemes are needed to efficiently manage the available cache space in order to reap higher gains. On average, the Loc-PF, LocWF-PF and LocWFCL-PF prefetchers achieve over 8.1%, 8.7% and 9.3% IPC improvements over the baseline system respectively, which is more than 33% of performance benefit achievable by making the L2 caches perfect. For streamcluster benchmark, which is significantly limited by memory performance and bandwidth, Loc-WF, LocWF-PF and LocWFCL-PF schemes provide as high as 56%, 58% and 60% performance gains respectively. Other benchmarks like nw, Triad and scalarProd which are also

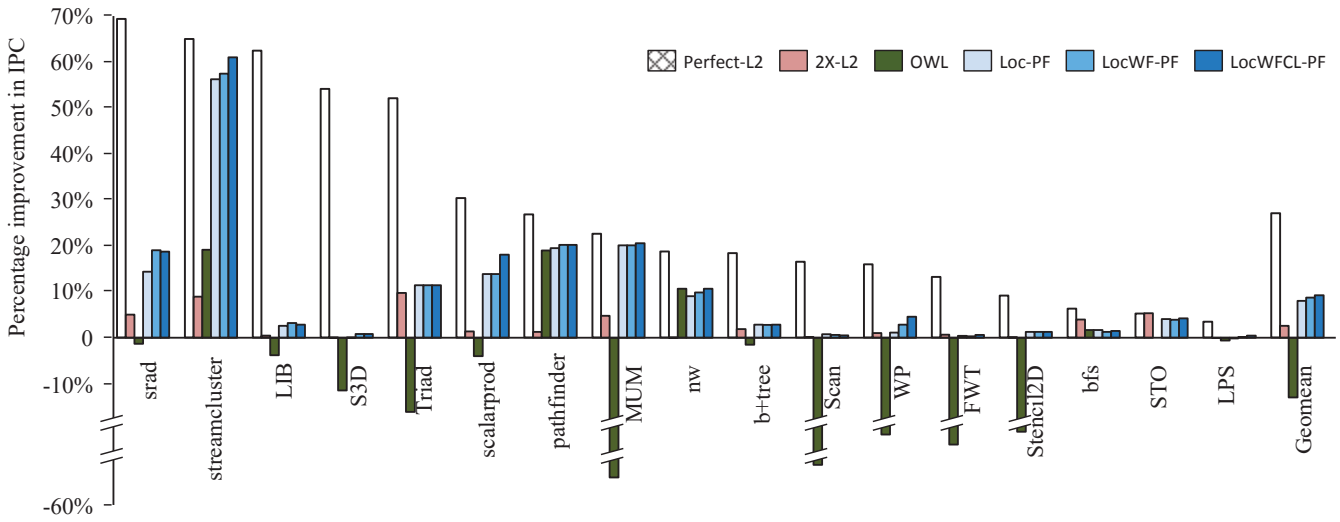

Figure 15: Instructions per cycle

limited by memory performance also get performance benefits up to ∼20% by prefetching. Bfs benchmark has limited temporal locality and exhibits streaming behavior with difficult-to-predict patterns for determining the neighboring search nodes. Thus, the row-based prefetchers improve the performance of bfs by ∼2.5% only. Also, benchmarks like bfs, b+tree and STO do not exhibit significant intra- or inter-wf correlation in row access patterns, thus, they suffer from marginal performance degradation with the LocWF-PF scheme as compared to the Loc-PF scheme. Benchmarks like scalarProd, streamcluster and nw have low cache-line re-usability and thus, they benefit much more from the LocWFCL-PF prefetching scheme.

The OWL prefetcher aggressively prefetches unaccessed cache lines in an open DRAM row into L2 just before the row is closed, thereby improving row buffer locality. It was originally evaluated with a coordinated wavefront-scheduling policy that prioritized scheduling of certain wavefront-groups to improve bank-level parallelism at the cost of row buffer locality. However, when evaluated with a more general and widely-used scheduling policy like GTO (and two other schedulers shown later) the OWL prefetcher reduces performance as compared to the baseline by over 10%. This is because if the scheduler allows more concurrently-active contexts and thus memory streams, OWL's aggressive prefetching policy causes significant cache pollution. Also, OWL naively triggers prefetches without considering the usefulness of the prefetched cachelines, and thus, can cause many unnecessary DRAM accesses. On the other hand, we will show that the effectiveness of the proposed prefetchers is independent of the front-end scheduling policy. Overall, the proposed prefetchers outperform both the OWL prefetcher and 2X-L2 configurations.

Figure 16 shows the impact of the proposed prefetchers on performance of the prefetch-agnostic benchmarks. These benchmarks are compute-intensive and can effectively hide their memory access latencies with available thread-level parallelism. Thus, they do not see much performance benefit ($\leq$ 3% IPC increase) with perfect-L2 caches. For these benchmarks, we can observe that the proposed prefetchers still lead to marginal performance gains.

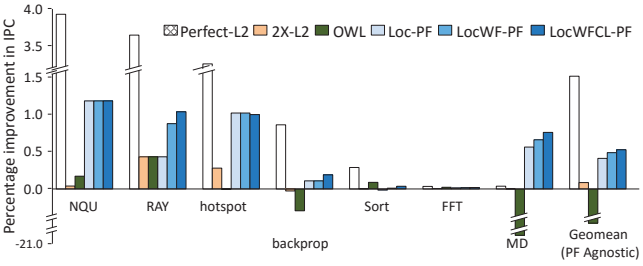

Figure 16: IPC for prefetch-agnostic benchmarks

5.2 Analysis of Performance

In this section, we analyze the performance implications of the proposed prefetching techniques using several key metrics.

5.2.1 Prefetcher accuracy and coverage

Figure 17 compares the coverage and accuracy of the three prefetching proposals across the prefetch-friendly benchmarks. Prefetcher accuracy is defined as the percentage of useful

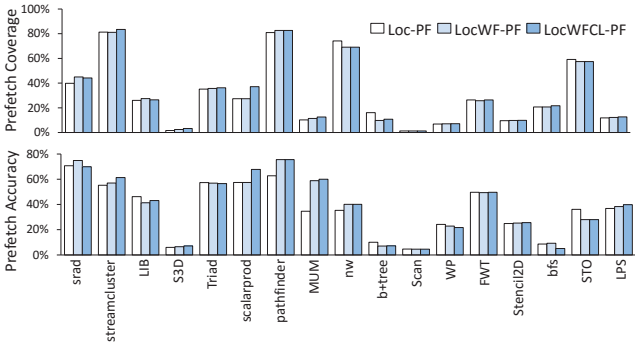

Figure 17: Prefetcher accuracy and coverage

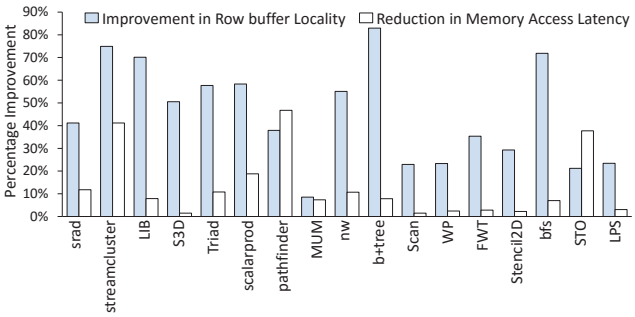

Figure 18: Improvement in row buffer locality and reduction in memory access latency

prefetches as compared to the total number of prefetches made and prefetcher coverage is defined as the percentage of demand requests that are satisfied by prefetches over all the demand requests. We can see that except a few benchmarks like S3D, b+tree, scan and bfs, all three prefetchers achieve good prefetch accuracies as high as 73%. This is because the schemes intelligently control the prefetching degree to minimize wasted prefetches and contention in the associated structures. It is interesting to note that even though FWT benchmark has good prefetch accuracy and coverage for all three proposals, it does not see as much overall performance benefit from prefetching due to the effect of write misses and dirty cacheline evictions. For applications with poor intra- and inter-WF locality properties like b+tree and bfs, the accuracy of the wavefront-correlation scheme is reduced as compared to the locality-based scheme, leading to lower performance gains.

5.2.2 Improvement in row buffer locality

A key advantage of the proposed prefetchers is that they leverage their proximity to the memory system and their understanding of the current state of the memory system to fetch data from DRAM row buffers into on-chip prefetch buffers before the actual demand requests arrive. Figure 18 shows the average improvement in row buffer locality (percentage of requests serviced from open row buffers over all memory requests) across prefetch-friendly benchmarks for the best-performing LocWFCL-PF scheme. We can clearly see that row buffer locality improves by over 38% by servicing prefetch requests to maximize row buffer hits.

5.2.3 Reduction in memory access latency

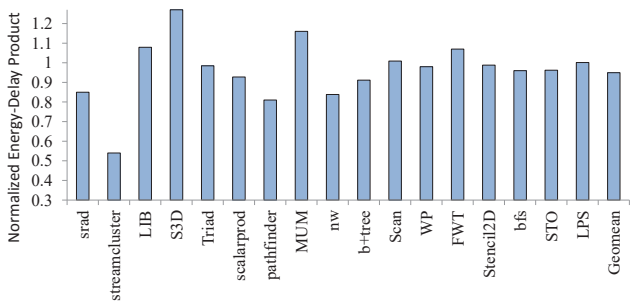

Figure 19: Energy-Delay product comparison

Figure 18 shows the reduction in average memory access latency (round-trip latency between core and memory) across the prefetch-friendly benchmarks for the best-performing LocWFCL-PF scheme. The proposed prefetching techniques leverage the lower-latency and energy-efficient access of row buffers for prefetching. Also, by focusing on maximizing the prefetch accuracy, the prefetcher avoids thrashing in the prefetch buffer and thus, reduces contention in the memory subsystem. As a result, the proposed techniques lead to significant reductions in memory access latencies by up to 40% (over 12% on average). Unlike other benchmarks, pathfinder and STO achieve higher reduction in memory access latencies than improvement in row buffer locality due to their high prefetch buffer hit rates.

5.3 Hardware Cost Analysis

Next, we evaluate the hardware overhead of the proposed prefetchers. The RTT table consists of 32-entries per memory controller (MC), with each entry being approximately 8 bytes, this makes the total hardware overhead of the RTT table to be roughly 2KB overall. Similarly, the WFT table also tracks 32 entries per MC, with each entry having an approximate size of 6 bytes, which makes the hardware overhead of the WFT table to be roughly 1.5KB overall. The GPT table consists of 64 entries, with 34-bits per entry, totaling up to 2.1KB overall. We also add a prefetch request queue per MC and the global timers, which together add up to less than 0.5KB of state in all. Thus, the LocWF-PF scheme adds a minimal additional state of approximately 2.5KB to the baseline PIM architecture, while wavefront-correlation adds another 3.6 KB of state.

5.4 Energy Efficiency Analysis

In this section, we analyze the impact of the prefetcher on energy-efficiency of the applications. We assume 20nm technology process for all logic. We use GPUWatch [33] to estimate the power of different processor configurations. We used CACTI to evaluate the power/energy overhead associated with the prefetcher predictor structures. The energy consumption of 3D-stacked DRAM is modeled using the parameters described in [13, 14].

Energy-Delay product (EDP) is a widely used metric that couples energy consumption and performance. Figure 19 shows the EDP of the prefetcher-enabled GPU-PIM system normalized over the no-prefetching configuration. Prefetching increases power consumption (by ∼7% on average) due to increased activity in the additional structures (RTT, WTT,

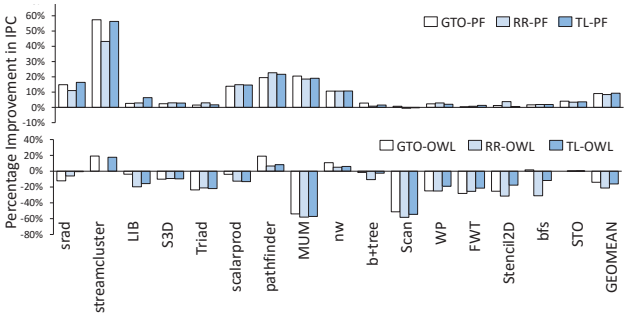

Figure 20: Sensitivity of performance to scheduling policy

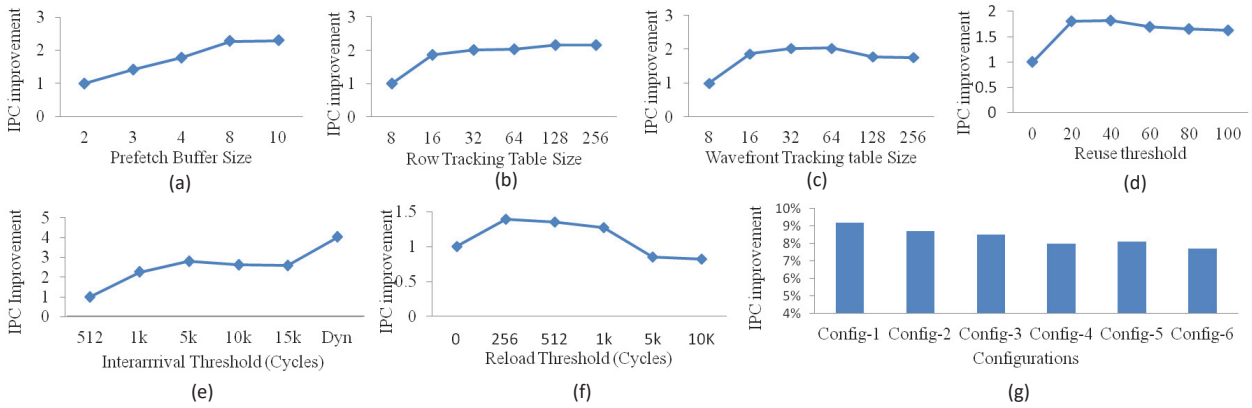

Figure 21: Sensitivity Studies

GPT) and increased number of prefetches to the memory system as compared to the baseline. But it also improves row buffer locality significantly, reduces contention in memory system due to higher prefetcher coverage and improves overall performance, thus reducing average energy by ∼3% and EDP of applications by over 5.1% as compared to the baseline. Overall, power used by our GPU-PIM design is small enough to be cooled using reasonable cooling solutions (e.g., commodity-server active heat sinks) [21].

5.5 Design-space Exploration

In this section, we perform design-space exploration by varying different prefetcher parameters.

5.5.1 Sensitivity to scheduling policy

Figure 20 shows the effect of three different wavefront-scheduling policies, Greedy-then Oldest (GTO), Round-robin (RR) and two-level scheduling (TL) on the performance of the best performing LocWFCL-PF prefetcher and the OWL prefetcher. We can see that the proposed prefetcher improves performance over the baseline no-prefetching configurations with GTO, RR and TL scheduling schemes by approximately 9%. The RR policy schedules consecutive wavefronts having higher locality characteristics in consecutive cycles and thus, it benefits comparatively less from prefetching. Nevertheless, the prefetcher exploits the synergy in the wavefront-localized memory access patterns to prefetch rows before actual demand requests, thereby still yielding over 8% performance improvement with the RR scheduling policy. It is important to note that the proposed prefetcher is effective irrespective of the scheduling policy while OWL's performance varies significantly with different scheduling policies. Also, OWL being an aggressive prefetcher deteriorates performance when used with all three widely-used schedulers as they do not restrict the number of concurrently active threads significantly.

5.5.2 Impact of prefetch buffer size

Figure 21a shows the performance impact of increasing the prefetch buffer (PB) size from 2 to 10 rows per memory controller, normalized over a PB size of 2. Performance improves by over 78% by increasing PB size from 2 to 4 entries and continues to increase significantly till 8 entries, beyond which the performance gains are marginal for many benchmarks. This is because row buffer locality improves significantly till 8 PB entries, but does not increase much beyond that. Also, prefetching more rows creates contention for demand requests in the memory system which limits the performance gains.

5.5.3 Impact of predictor table size

Figure 21b shows the impact of RTT size on performance as it increases from 8 to 256 entries (per MC), normalized with respect to an 8-entry RTT. Performance improves significantly up to 32 RTT entries. Tracking fewer entries in the RTT table creates contention and leads to sub-optimal next-row prefetching decisions as the pool of candidate rows is smaller. On the other hand, tracking large number of RTT entries (beyond 128) does not provide much increased benefit because of diminishing returns from selecting a row from a pool of 256 candidates. In some cases, it also degrades performance because it increases the likelihood of prefetching a "stale" tracked row into the prefetch buffer. Using up to 32 entries in the RTT yields good performance gains.

Figure 21c shows the impact of WFT size on performance as it increases from 8 to 256 entries (per MC), normalized with respect to an 8-entry WFT. Using up to 64 entries yields the highest performance gains. This is because tracking and prefetching for more wavefronts creates aliasing and confusion in the delta-pattern tables and also leads to more aggressive prefetching which reduces performance gains.

5.5.4 Impact of prefetch buffer management thresholds

Figure 21d shows the impact of varying the reuse-threshold on performance. Too low a reuse threshold implies that the applications mostly operate in low-reuse mode and thus, it hurts performance of applications that have high re-usability properties and vice versa. A reuse-threshold value of 30-40% yields the best performance gains.

Figure 21e shows the impact of varying inter-arrival time threshold on performance, normalized with respect to a fixed threshold of 500 cycles. Choosing an appropriate inter-arrival threshold is crucial, smaller threshold leads to premature eviction of active rows from the PB and larger threshold causes rows to be cached in the PB for a long time without

providing any benefit. Using a fixed inter-arrival threshold of 1K (lower) cycles degrades performance of benchmarks such as b+tree, mummerGPU, etc., which have higher inter-arrival time on average. Using a fixed inter-arrival threshold of 15K (high) cycles reduces performance gains for most benchmarks such as srad, streamcluster, Triad, etc. that have lower inter-arrival times. The dynamic threshold selection scheme (Dyn) provides best performance gains by selecting the threshold dynamically based on application behavior.

Figure 21f shows the impact of different reload-time thresholds on performance, normalized over the performance with a fixed reload threshold of zero cycles. We can see that a reload threshold of 256 cycles provides the best performance as compared to other values as it prevents premature eviction of the "all-lines" touched rows for benchmarks that have a higher likelihood of re-using their prefetched rows, while not impacting the performance for other benchmarks.

5.5.5 Effect of varying the CU count and L2 size

Figure 21g shows the performance improvement achieved with different GPU-PIM configurations. Configs 1-3 employ a 128 KB L2 cache with 4, 6 and 8 CUs respectively. Configs 4 and 5 use a 256 KB L2 cache with 6 and 8 CUs respectively while config 6 uses a 512 KB L2 cache with 10 CUs. Adding prefetching support improves performance across configs 1-5 by over 8-9%, while config 6 achieves over 7.5% performance improvement. This experiment demonstrates that the proposed prefetchers are effective at improving the performance of in-memory GPU processors over many configurations.

6. PRIOR WORK

Processing-in-memory - Several research studies [42, 19, 41, 12, 22] have explored integrating logic and DRAM since the 1990s. Recent advances in 3D stacking [4, 5, 6] have renewed a lot of interest in PIM systems. Pugsley *et al.* proposed using a daisy-chain of HMC memory devices with simple logic cores for improving performance and energy-efficiency of MapReduce workloads [43]. The NDA architecture proposed stacking reconfigurable arrays on commodity DRAM devices [23]. In TOP-PIM [50], a 3D-stacked model with GPUs as PIM cores for HPC and graph workloads was proposed. Several other recent studies [47, 8, 7, 11, 51, 38] have also provided useful insights in PIM-augmented 3D-DRAM systems.

GPU Prefetching - Many researchers have studied GPU prefetching in the past. Lee *et al.* [32] proposed MTA, a core-side L1 prefetcher that used earlier threads to prefetch data for later threads so as to reduce cold misses. Jog *et al.* [25] proposed a prefetch-aware scheduling policy that controls scheduling of consecutive wavefronts to allow wavefronts to prefetch for each other and a cooperative, simple core-side prefetcher that prefetches from hot memory-regions after a threshold number of cachelines within the region get accessed. Jog *et al.* [26] also proposed OWL, a memory-side prefetcher for GPUs that prefetches unaccessed cachelines from a DRAM row before closing it in order to improve row buffer locality. We compare our memory-side prefetching proposals extensively with the OWL prefetcher. Apogee [44], another core-side prefetcher leverages fixed-offset address and thread invariant access patterns combined with a prefetch-throttling mechanism. Lakshminarayana *et al.* [31] proposed prefetching into the register file for several graph applications.

CPU Prefetching - Hardware prefetching has been extensively studied in CPUs [20, 27, 40, 46, 45]. Srinath *et al.* [46] proposed using feedback mechanisms to dynamically adjust the desired prefetcher accuracy and timeliness in CPUs. Nesbit *et al.* [40] proposed the Global History Buffer (GHB) to improve prefetcher effectiveness and reduce table sizes. Somogyi *et al.* [45] correlate spatial locality in the accessed memory regions with the program counter data to identify prefetch candidates for commercial workloads.

7. CONCLUSION

In this paper, we proposed three light-weight, practical memory-side prefetcher designs, which improve the performance and energy efficiency of GPU-PIM systems. The first proposal, locality-aware prefetcher exploits the lower-latency and energy-efficient access of DRAM row buffers coupled with the application's past access history to make prefetch decisions. In order to maximize the utilization of prefetched data and minimize thrashing effects, the locality-aware prefetcher combines a novel prefetch buffer management policy based on a unique dead-row prediction mechanism and an eviction-based prefetch-trigger policy to control its aggressiveness. The second proposal, wavefront-correlation-aware prefetcher builds upon the first heuristic and exploits the synergy and regularity in the wavefront-localized memory access streams to improve prefetcher accuracy and timeliness. Finally, we proposed cacheline-reuse-aware prefetcher, as an enhancement over the previous two heuristics, which leverages application's temporal re-usability characteristics to maximize utilization of prefetch buffer space. The proposed prefetchers improve overall performance by over 60% (maximum) and 9% (average) as compared to the baseline system, while achieving over 33% of the performance benefits of perfect-L2 system. Moreover, the proposed prefetchers outperform the state-of-the-art memory-side prefetcher, OWL by more than 20%.

8. ACKNOWLEDGMENT

The authors of this work are supported partially by SRC under Task ID 2504, National Science Foundation (NSF) under grant number 1337393 and AMD. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other sponsors. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

9. REFERENCES

[1] Advancing Moore's Law on 2014! http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf.
[2] NVIDIA's next generation CUDA compute architecture, Fermi, 2009.
[3] Nvidia. CUDA c/c++ sdk code samples, 2011.
[4] Hybrid memory cube consortium. Hybrid Memory Cube Specification 1.0, 2013.
[5] Jedec standard jesd235. High Bandwidth Memory (HBM) DRAM, 2013.

[6] Jedec standard jesd235a. High Bandwidth Memory (HBM) 2 DRAM, 2016.

[7] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[8] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[9] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *PPoPP*, pages 23–34, New York, NY, USA, 2012. ACM.

[10] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174. IEEE Computer Society, 2009.

[11] P. Balaprakash, D. Buntinas, A. Chan, A. Guha, R. Gupta, S. H. K. Narayanan, A. A. Chien, P. Hovland, and B. Norris. Exascale workload characterization and architecture implications. In *Proceedings of the High Performance Computing Symposium*, HPC '13, pages 5:1–5:8, San Diego, CA, USA, 2013. Society for Computer Simulation International.

[12] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: building a smarter memory controller. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 70–79, Jan. 1999.

[13] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens. System and circuit level power modeling of energy-efficient 3d-stacked wide i/o drams. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 236–241, San Jose, CA, USA, 2013. EDA Consortium.

[14] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. Drampower: Open-source dram power and energy estimation tool.

[15] D. Chang, G. Byun, H. Kim, M. Ahn, S. Ryu, N. Kim, and M. Schulte. Reevaluating the latency claims of 3d stacked memories. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 657–662, Jan 2013.

[16] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IISWC*, pages 185–195. IEEE Computer Society, 2013.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[18] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU-3*, pages 63–74, New York, NY, USA, 2010. ACM.

[19] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 14–25, New York, NY, USA, 2002. ACM.

[20] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, pages 7–17. IEEE Computer Society, 2009.

[21] Y. Eckert, N. Jayasena, and G. H. Loh. Thermal feasibility of die-stacked processing in memory. In *WoNDP: 2nd Workshop on Near-Data Processing*, 2014.

[22] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 232–241, New York, NY, USA, 2007. ACM.

[23] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295, Feb 2015.

[24] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 209–220, Washington, DC, USA, 2002. IEEE Computer Society.

[25] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[26] A. Jog, O. Kayiran, K. Mishra, and M. T. K. Owl: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *In ASPLOS*, 2013.

[27] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 252–263, New York, NY, USA, 1997. ACM.

[28] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, 2013.

[29] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *MICRO*, pages 175–186, Washington, DC, USA, 2010. IEEE Computer Society.

[30] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block

prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 144–154, New York, NY, USA, 2001. ACM.

[31] N. B. Lakshminarayana and H. Kim. Spare register aware prefetching for graph algorithms on GPUs. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 614–625, 2014.

[32] J. Lee, N. B. Lakshminarayana, H. Kim, and R. W. Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *MICRO*, pages 213–224. IEEE Computer Society, 2010.

[33] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 487–498, New York, NY, USA, 2013. ACM.

[34] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 67–77, New York, NY, USA, 2015. ACM.

[35] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based cache allocation in throughput processors. In *HPCA*, pages 89–100. IEEE, 2015.

[36] K.-N. Lim, W.-J. Jang, H.-S. Won, K.-Y. Lee, H. Kim, D.-W. Kim, M.-H. Cho, S.-L. Kim, J.-H. Kang, K.-W. Park, and B.-T. Jeong. A 1.2v 23nm 6f2 4gb ddr3 sdram with local-bitline sense amplifier, hybrid lio sense amplifier and dummy-less array architecture. In *ISSCC*, pages 42–44. IEEE, 2012.

[37] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 222–233, 2008.

[38] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski. A processing in memory taxonomy and a case for studying fixed-function pim. In *WoNDP: 1st Workshop on Near-Data Processing*, 2013.

[39] S. Mu, Y. Deng, Y. Chen, H. Li, J. Pan, W. Zhang, and Z. Wang. Orchestrating cache management and memory scheduling for GPGPU applications. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(8):1803–1814, Aug 2014.

[40] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 96–, Washington, DC, USA, 2004. IEEE Computer Society.

[41] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *ISCA*, pages 192–203, Washington, DC, USA, 1998. IEEE Computer Society.

[42] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, Mar. 1997.

[43] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, 0:190–200, 2014.

[44] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. Apogee: Adaptive prefetching on GPUs for energy efficiency. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 73–82, Piscataway, NJ, USA, 2013. IEEE Press.

[45] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society.

[46] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, Washington, DC, USA, 2007.

[47] J. Torrellas. Flexram: Toward an advanced intelligent memory system: A retrospective paper. In *ICCD*, pages 3–4. IEEE Computer Society, 2012.

[48] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246. IEEE Computer Society, 2010.

[49] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[50] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 85–98, New York, NY, USA, 2014.

[51] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, pages 7:1–7:3, New York, NY, USA, 2013. ACM.