Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities

Ashutosh Pattnaik[1] Xulong Tang[1] Adwait Jog[2] Onur Kayıran[3]
Asit K. Mishra[4] Mahmut T. Kandemir[1] Onur Mutlu[5,6] Chita R. Das[1]

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

ABSTRACT

Processing data in or near memory (PIM), as opposed to in conventional computational units in a processor, can greatly alleviate the performance and energy penalties of data transfers from/to main memory. Graphics Processing Unit (GPU) architectures and applications, where main memory bandwidth is a critical bottleneck, can benefit from the use of PIM. To this end, an application should be properly partitioned and scheduled to execute on either the main, powerful GPU cores that are far away from memory or the auxiliary, simple GPU cores that are close to memory (e.g., in the logic layer of 3D-stacked DRAM).

This paper investigates two key code scheduling issues in such a GPU architecture that has PIM capabilities, to maximize performance and energy-efficiency: (1) how to automatically identify the code segments, or kernels, to be offloaded to the cores in memory, and (2) how to concurrently schedule multiple kernels on the main GPU cores and the auxiliary GPU cores in memory. We develop two new runtime techniques: (1) a *regression-based affinity prediction model and mechanism* that accurately identifies which kernels would benefit from PIM and offloads them to GPU cores in memory, and (2) a *concurrent kernel management* mechanism that uses the affinity prediction model, a new kernel execution time prediction model, and kernel dependency information to decide which kernels to schedule concurrently on main GPU cores and the GPU cores in memory. Our experimental evaluations across 25 GPU applications demonstrate that these two techniques can significantly improve both application performance (by 25% and 42%, respectively, on average) and energy efficiency (by 28% and 27%).

1. INTRODUCTION

Graphics Processing Units (GPUs) provide very high computational bandwidth at a competitive power budget. These characteristics have led to their deployment in a wide range of platforms, including in many machines that appear in the Top500 and Green500 lists [1, 2]. Although GPUs are likely to play a promising role in the design of exascale systems, continuous scaling of their performance and energy efficiency will not be an easy task. One of the biggest impediments to this continuous scaling is the memory system energy consumption due to data transfer overhead [58]. A typical 64-bit DRAM access consumes about 100-1000X the energy consumed by a double-precision floating point operation [24,58], and this gap could increase with technology scaling [77]. Even with optimistic assumptions about improvements in memory technology, reducing the total DRAM access energy from approximately 18-22 pJ/b (in modern GDDR5) to 4 pJ/b and sustaining it for over 100,000 nodes, memory can still consume a significant fraction (e.g., 70%) of the total system's power budget [112].

Figure 1 illustrates the data movement and energy consumption overheads of transferring data between memory and computational units across 25 applications in a modern GPU system,[1] by showing: (1) the fraction of all data movement in the system that is due to off-chip transactions between memory and the GPU, and (2) the fraction of total system energy consumption that is due to this off-chip data movement. We observe that memory accesses result in 49% of all data movement and are responsible for 41% of the energy consumption of the system.

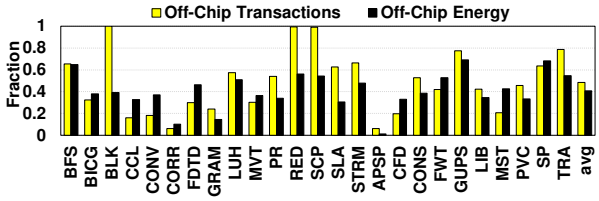


Figure 1: Data movement and system energy consumption caused by off-chip memory accesses.

Figure 2 shows the performance loss due to the overhead of transferring data from/to main memory, by plotting the normalized performance of our applications compared to an *idealized* system where all off-chip memory requests in our baseline are *magically* eliminated, i.e., forced to hit in the last-level cache.[2] Averaged across 25 applications, main memory accesses lead to 45% performance degradation.

A promising approach to minimize data movement, energy and performance overheads of main memory accesses is to move memory-intensive computations closer to memory, i.e., Processing-In Memory (PIM) [3,4,27,33], also known as Processing-Near Memory (PNM) or Near-Data Computing (NDC) [13]. The core of the PIM concept is to have computational units that are closely integrated with memory

PACT '16, September 11-15, 2016, Haifa, Israel

DOI: http://dx.doi.org/10.1145/2967938.2967940

[1]Section 6 provides our experimental methodology.

[2]The minimum DRAM latency after the last-level cache is 100 cycles (see Section 6 for details). The average latency is higher due to significant contention observed in the DRAM system [10, 26, 54, 57, 59–61, 63, 75, 76, 78, 79, 100–102, 107, 108].

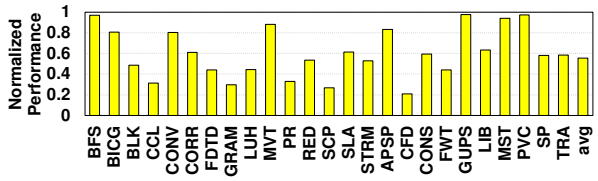


Figure 2: Performance normalized to a hypothetical GPU where all the off-chip accesses hit in the last-level cache.

such that data can be moved from memory to those units at much higher bandwidth, lower latency and lower energy than doable from memory to the main processor. While the PIM concept goes back to late 1960s [99] and it had gathered some momentum through several projects in 1990s (e.g. [33, 40, 55, 62, 67, 85]), its main technological limitation has been the difficulty of integrating computational units very close to memory. With the significant advances in adoption of 3D-stacked memory technology that tightly combines a logic layer and DRAM layers [3, 4, 48, 64, 71, 86, 109], this limitation has been overcome and PIM has become a likely-viable approach to improve system design.

The PIM approach has recently been explored (e.g., [44, 112]) for reducing memory bandwidth and minimizing the data transfer overheads between GPU cores and off-chip DRAM. A promising way to integrate PIM to a GPU-based system is what we call a *PIM-Assisted GPU architecture*, where at least one 3D-stacked memory chip is placed adjacent to the GPU chip, and both chips are interconnected via a memory link on an interposer (as depicted in Figure 3).[3] The 3D-stacked memory contains a base logic layer, housing GPU cores. This architecture has two types of compute engines: (1) large and powerful primary GPU cores, called GPU-PIC, i.e., processing-in-core, which are similar to modern GPU cores, and (2) smaller and less powerful GPU cores, called GPU-PIM, which are placed in the logic layer under main memory and assist in performing computation. No prior work explored how to fully exploit this architecture such that appropriate parts of an application are identified and scheduled to utilize *both* the main GPU cores (GPU-PIC) and cores in memory (GPU-PIM) to maximize the performance and energy-efficiency of the entire system.

Our goal in this paper is to develop mechanisms to fully and automatically exploit the performance and energy-efficiency potential of PIM-Assisted GPU architectures. To this end, we investigate **two key code scheduling problems**. First, *how to automatically identify the code segments to be offloaded to the GPU cores in memory.* Second, *how to concurrently schedule multiple kernels on the main GPU and the cores in memory.* To address these problems, one must consider several questions such as (1) what the granularity of the code segment that is offloaded to GPU-PIM should be, (2) how to determine which code segments benefit from being executed on GPU-PIM, (3) how to efficiently distribute work between the main GPU and the PIM engine to maximize system performance, (4) while executing each code segment on its preferred cores as much as possible.

To this end, we first characterize the execution behavior of different applications at the **kernel granularity** to estimate the performance and energy benefits when each individual kernel is placed in the main GPU or GPU-PIM. Because the CPU offloads computation to the GPU at the kernel-granularity, maintaining the same granularity for PIM execution enables low-overhead computation offloading from the CPU to both GPU-PIC and GPU-PIM. Based on this insight, we propose **two new runtime techniques**, the primary contributions of this paper, that address the two scheduling problems.

[3]Section 2.2 discusses details and advantages of this architecture.

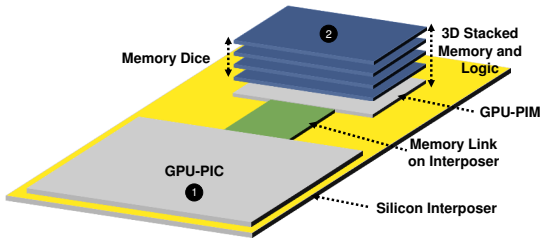


Figure 3: PIM-assisted GPU Architecture.

Kernel Offloading. As an application exhibits varying computation and memory demands during different phases of execution, some kernels (e.g., those that fit in the main GPU scratchpad) benefit more from executing on the main GPU, GPU-PIC, and others (e.g., those that overwhelm the memory bandwidth to the main GPU) on the GPU in memory, GPU-PIM. Thus, identifying the affinity of the kernels correctly and scheduling each on the appropriate computational engine would improve performance. To solve this problem, we develop a *regression-based affinity prediction model and mechanism* that accurately identifies, at runtime, which kernels would benefit from executing on PIM cores and offloads them to the GPU cores in memory. Our regression model, which is built on an in-depth kernel level analysis, considers three broad kernel-level metrics (memory intensity, kernel parallelism, and shared memory (software managed scratchpad) intensity) and is trained using applications randomly picked from our pool of 25 applications (i.e., the training set). Our detailed experimental evaluations on the remaining applications (i.e., the test set) show that the proposed mechanism improves average performance by 25% and energy efficiency by 28% compared to a baseline conventional GPU architecture that contains the same number of GPU cores as that of the combined number of GPU cores present in both GPU-PIC and GPU-PIM.

Concurrent Kernel Management. We find that even a highly accurate prediction mechanism for kernel offloading can leave many resources (e.g., GPU-PIC) underutilized. During the execution of a kernel on either the main GPU or GPU-PIM, the other device is left unutilized, which limits achievable system performance. Thus, identifying *independent kernels* that can be scheduled concurrently and scheduling them on GPU-PIC and GPU-PIM at the same time, in a manner that minimizes overall application execution time, can significantly improve system performance and efficiency. To solve this problem, we develop a *concurrent kernel management* mechanism that uses the affinity prediction model, a new regression-based kernel execution time prediction model, and dependency information across kernels to decide which kernels to schedule concurrently on the main GPU cores and the GPU cores in memory. Our detailed experimental evaluations indicate that our technique improves average performance by 42% and energy-efficiency by 27%, compared to the same baseline described above.

This paper makes the following major **contributions**:

1. It provides an in-depth kernel-level analysis of GPU application behavior with respect to suitability for Processing in Memory. It makes an experimentally-supported case for the use of *kernel granularity* to identify and schedule GPU

program code segments to execute on either the main GPU cores or the GPU cores in memory.

2. It develops a new regression-based affinity prediction model to estimate the best compute engine to execute a kernel in a PIM-assisted GPU architecture. We show how to use this model to guide a new *kernel offloading* mechanism to GPU cores in memory.

3. It develops a new execution time prediction model for kernel execution in a PIM-assisted GPU architecture. We show how to use this model to guide a new *concurrent kernel management* mechanism that executes multiple kernels concurrently in the main GPU and the in-memory GPU cores in a PIM-assisted GPU architecture.

4. It comprehensively evaluates the *kernel offloading* and *concurrent kernel management* mechanisms and shows that both improve performance and energy efficiency significantly while requiring no support from the programmer.

2. BACKGROUND

This section provides a brief background on GPUs and the PIM-assisted GPU architecture.

2.1 Conventional GPU Architectures

Our baseline conventional GPU consists of multiple cores, also called streaming-multiprocessors (SMs)[4] in NVIDIA terminology [82]. Each SM has a private L1 data cache, a texture cache and a constant cache, along with a software-managed scratchpad memory (called *shared memory*). The SMs are connected to memory channels (partitions) via an interconnection network. Each memory partition is directly connected to a partition of the L2 cache, which is shared by all the cores, and a memory controller. The memory requests are buffered and scheduled by the memory controllers [9, 54, 106]. There are multiple memory controllers, each controlling a memory partition. Data is interleaved at the chunk granularity across the controllers. The parallel parts of a CUDA/OpenCL application, which are offloaded to the GPU, are called *kernels*. The execution of an application starts with the launch of a kernel on the GPU. The kernel launch involves copying the kernel code and the data from the CPU memory to the GPU memory. Once the kernel finishes execution on the GPU, its results are copied back to the CPU memory from the GPU memory.

2.2 PIM-Assisted GPU Architectures

As we discussed in Section 1, 3D-stacked memory technology enables the ability to place computational units in the base logic layer that is underneath the memory stacks [3, 4, 48, 64, 71, 86, 109]. There can be multiple strategies to exploit such a PIM technology in a GPU system.

One strategy is to stack the memory directly on top of a conventional GPU architecture, by placing a conventional GPU in the logic layer [114]. Although such an organization provides the benefits of tight GPU and memory coupling such as high bandwidth and low access latency, it has two primary issues. First, heat from the processor might significantly degrade the retention time of the 3D-stacked memory [68]. Consequently, the refresh rate of the DRAM might need to be increased, leading to reduced peak performance and higher energy consumption [20, 53, 68, 69, 88]. Second, such an organization limits the total memory capacity that could be stacked within the area of the processor.

[4]In this paper, we use the terms *core* and *SM* interchangeably.

Another strategy [44,112] to exploit PIM in a GPU system is to keep the main GPU same as in conventional systems but connect to it, via memory links on a silicon interposer, one or more 3D-stacked memory units that are capable of doing computation, as depicted in Figure 3 with one 3D-stacked memory. The base logic layer of each 3D-stacked memory houses an auxiliary GPU that is simpler and more power-efficient than the main GPU. Such an organization has considerably lower thermal constraints than the previous strategy and is more scalable in terms of memory capacity. However, if computations are not scheduled appropriately across the main GPU and the computational units in the 3D-stacked memories, the energy consumption, latency and bandwidth overheads of the interposer may limit performance due to excessive communication between the main GPU and the 3D-memory stacks.

In this paper, we call the latter organization (of Figure 3) the *PIM-Assisted GPU architecture*, and aim to maximize its benefits by scheduling computations intelligently across the main GPU and PIM units. This architecture is a heterogeneous system. The main GPU chip, which we call the processing-in-core architecture (GPU-PIC ❶ in Figure 3), provides high throughput to compute-intensive GPGPU applications, but it has limited memory bandwidth due to its horizontal integration with the memory stack (bottlenecked by the memory links). On the other hand, the PIM cores on the base logic layer of the 3D memory stack, which we call the processing-in-memory architecture (GPU-PIM ❷ in Figure 3), achieves the full bandwidth and energy efficiency of 3D stacking of memory and logic, but provides a peak instruction throughput lower than that of GPU-PIC due to the smaller number of the execution engines in the logic layer. Therefore, placing computation correctly on these two different types of GPU units, GPU-PIC and GPU-PIM, is critical for performance and energy efficiency. For example, computations that are memory-intensive and can tolerate the lower parallelism present in the logic layer of GPUs are likely better executed on GPU-PIM instead of GPU-PIC.

Unlike GPU-PIC, GPU-PIM has a direct interface to 3D-stacked DRAM. Therefore, GPU-PIM is able to sustain much higher memory bandwidth (in our configuration, 4×) and it experiences much lower memory latency than GPU-PIC. Because of this, we observe that GPU-PIM does *not* significantly benefit from having an L2 cache and we evaluate a GPU-PIC design without an L2 cache. Table 4 provides the details of our GPU-PIC and GPU-PIM configuration, which is similar to the TOP-PIM configuration [112]. We also perform a sensitivity analysis in Section 8 on the number of cores and cache in GPU-PIM. Note that the cores in GPU-PIC and GPU-PIM use the same ISA, and thus, have the same programmability features.

Thermal Feasibility. Eckert et al. [29] provide extensive models to demonstrate the thermal feasibility of PIM-based GPU architectures. They argue that the power consumed by the logic layer at the GPU-PIM at an ambient temperature of 30°C can be as much as 50W and describe that this is thermally feasible. Considering their study and configuration parameters from [112], our GPU-PIM has four times fewer SIMD compute units, similar to [44]. We estimate the maximum chip power usage of GPU-PIM when running MaxFlops [25][5] to be approximately 45W, using GPUWattch [65]. GPUWattch models NVIDIA Fermi [82]

[5]MaxFlops is compute-intensive benchmark that exhibits high dynamic core power consumption.

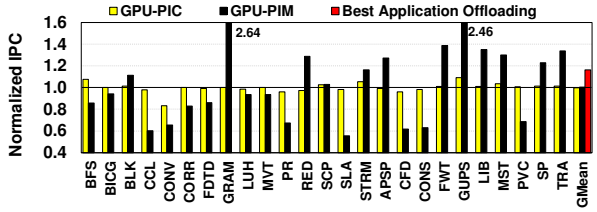
SMs, which are obsolete and not power-efficient compared to current generation GPU cores [83]. Therefore, we believe GPU-PIM will be even more power efficient when fabricated for state-of-the-art and future power-efficient designs.

3. MOTIVATION

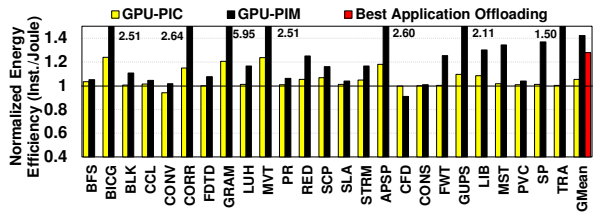
The PIM-assisted GPU architecture is a scalable and heterogeneous substrate. It provides the flexibility of adding more computational units on the GPU-PIC, and more memory capacity and bandwidth by incorporating additional 3D memory stacks containing GPU-PIM. At the same time, GPU-PIC and GPU-PIM architectures are quite different in terms of their ability to cater to applications with varying computation and memory demands. Given such heterogeneity in the PIM-assisted GPU architecture, it might be desirable for the CPU to offload a compute-bound GPGPU application onto GPU-PIC and a memory-bound GPGPU application onto GPU-PIM. However, this is not optimal, as we demonstrate in this section since it does not fully utilize both the GPU-PIC and GPU-PIM. There are many challenges in designing an offloading strategy to *maximize overall application performance and energy efficiency* by appropriately partitioning and scheduling an application across GPU-PIC and GPU-PIM. This section motivates the potential opportunities and limitations of *application offloading* to motivate our approach of *kernel offloading*.

3.1 Benefits of Application Offloading

Figure 4 shows the normalized performance (in terms of IPC) and energy efficiency (in terms of Instructions/Joules), compared to our baseline conventional GPU architecture with equivalent number of computation units (40 cores), when each application from our workload suite are offloaded by the CPU on either the GPU-PIM (8 cores) or the GPU-PIC (32 cores) in our PIM-Assisted GPU Architecture. *Best Application Offloading* shows the average performance and energy efficiency when each application is, with ideal knowledge, offloaded to the computation unit that provides the *best performance* for that application. We make three major observations.



(a) Performance.



(b) Energy Efficiency.

Figure 4: Effect of application offloading.[6]

[6]The results are normalized to a conventional GPU which has the combined peak instruction throughput of GPU-PIC and GPU-PIM (i.e., 40 cores). The entire application is offloaded to either GPU-PIC (32 cores) or GPU-PIM (8 cores).

First, offloading all applications to either GPU-PIC or GPU-PIM yields similar average performance because some applications prefer GPU-PIC and others GPU-PIM. We find that many applications significantly gain from the high memory bandwidth and low memory latency benefits of GPU-PIM. Some applications, e.g., `GRAM` and `APSP`, benefit from GPU-PIM due to low memory access latencies. These two applications (`GRAM` and `APSP`) have limited parallelism (as shown later in Table 2), leading to poor latency tolerance. Neither of these applications stress the memory bandwidth and they have a small fraction of accesses going to off-chip memory (Figure 1). In contrast, many applications, such as `CCL`, `PR`, and `CFD`, experience performance degradation when executed on GPU-PIM, because they 1) are bottlenecked by the limited computational power of the PIM cores, 2) do not effectively utilize the high bandwidth of GPU-PIM because they generate a small number of memory requests.

Second, offloading all applications to GPU-PIM provides the highest average energy efficiency (Instructions/Joule), even higher than the *Best Application Offloading* mechanism which chooses the unit that provides the best *performance* on a per-application basis. This is because GPU-PIM leads to a much more energy-efficient memory system, on average. Applications such as `GUPS`, `GRAM`, and `APSP` benefit significantly from GPU-PIM in terms of energy efficiency, primarily because of the reduced execution time due to the performance improvements.

Third, an optimal application offloading scheme that can detect the best platform to execute an application on, based on performance, can provide both performance and energy improvements than offloading always to either GPU-PIC or GPU-PIM. On average, GPU-PIM improves energy efficiency by 42% over GPU-PIC while having the same performance of GPU-PIC. The optimal application offloading scheme (in terms of performance) improves performance by 16% and energy efficiency by 28% over the baseline. Note that the optimal scheme is optimized for performance, not energy efficiency, and therefore its energy efficiency is less than that of offloading all applications to GPU-PIM.

We also note that an optimal offloading scheme could be different based on the metric to optimize for. For example, for `MVT`, GPU-PIM is much more energy-efficient but lower performance than GPU-PIC. There is no clear winner as one could optimize for either performance or energy-efficiency, depending on the use-case. In this paper, we focus on optimizing performance. However, we also demonstrate the positive impact of our schemes on energy efficiency.

3.2 Limitations of Application Offloading

We demonstrate the limitations of application offloading. Even for an optimal *application* offloading strategy, we find two major limitations that need to be addressed to make offloading more efficient in terms of performance.

Limitation I: Lack of Fine-Grained Offloading. We observe that offloading at the granularity of each application is too coarse-grained to take advantage of the different characteristics of code present within an application that can favor either GPU-PIC or GPU-PIM. To motivate this, Figure 5 shows a *kernel-level*, i.e., finer-grained, execution time breakdown of four representative GPGPU applications on GPU-PIC and GPU-PIM, normalized to the execution time on GPU-PIC. We show only the kernels that dominate each application's execution time. Three observations are in

order. First, CONS has two representative kernels and execution times of both kernels increase on GPU-PIM. Therefore, GPU-PIM is not an appropriate computation unit for any of CONS's kernels. Second, FWT has four representative kernels and execution times of all kernels decrease on GPU-PIM. Therefore, GPU-PIM is an appropriate configuration for all its kernels. Third, LUH and FDTD have kernels that demonstrate different behavior. Although both applications as a whole have higher execution times on GPU-PIM, some of their kernels actually benefit from GPU-PIM. For example, in LUH and FDTD, Kernel-K1 has lower execution time on GPU-PIM compared to GPU-PIC.

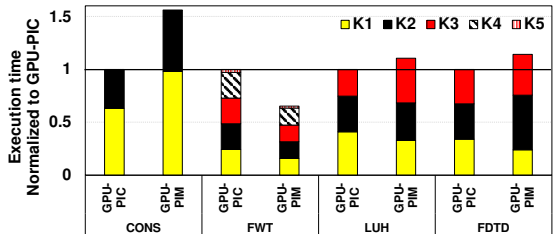


Figure 5: Breakdown of the execution time across different kernels for four representative GPGPU applications.

Thus, a careful *kernel-level* offloading strategy can perform even better than the application offloading strategy we evaluated (Section 3.1). Figure 6 illustrates the potential of kernel offloading for FDTD. Scenario-I and Scenario-II are the two possible application offloading strategies adopted to execute the entire FDTD on GPU-PIM or GPU-PIC, respectively. In the kernel offloading strategy (Scenario-III), each *kernel* of FDTD is offloaded to the computation engine where its execution time is lower (i.e., each kernel is offloaded to the unit it has *affinity* towards). Therefore, Kernel-K1 is offloaded to GPU-PIM and the other two kernels are offloaded to GPU-PIC. Kernel offloading saves many execution cycles (A) even over the best application offloading strategy. However, the key challenge of kernel offloading is in identifying the *affinity* of each kernel, which we provide a new solution for in this paper (Section 4).

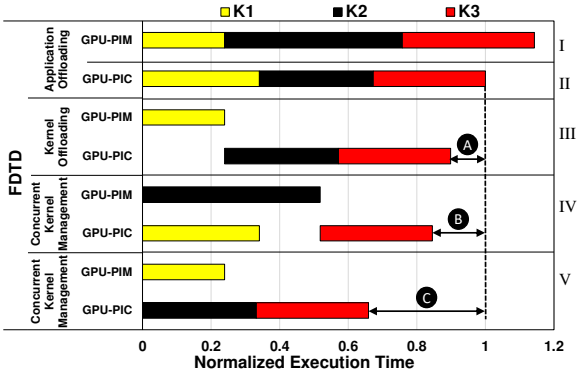


Figure 6: Performance advantages of kernel offloading (III) and concurrent kernel management (IV and V) mechanisms using the FDTD application as an example.

Limitation II: Lack of Concurrent Utilization of GPU-PIM and GPU-PIC. An application offloading mechanism loses out on the opportunity of using *both* the GPU-PIM and GPU-PIC at the same time, because it is too coarse-grained. In contrast, if offloading is performed at the kernel level, as described before, both GPU-PIM and GPU-PIC can be utilized by *concurrently* executing *independent* kernels on them. Figure 6 illustrates an example of this in Scenarios IV and V. In FDTD, Kernel-K3 can start executing only when both Kernel-K1 and Kernel-K2 finish their executions. However, Kernel-K1 and Kernel-K2 can execute in parallel. Because of the concurrent execution of kernels, overall application execution time is reduced (B) over the best application-offloading scenario, in scenario-IV. Scenario-V in Figure 6 demonstrates that kernel affinity, i.e., scheduling each kernel on the execution engine that is best for the kernel's performance matters: Kernels K1 and K2 respectively have affinity for GPU-PIM and GPU-PIC, and executing them concurrently on these engines leads to even higher overall application execution time savings (C) than in Scenario-IV where the same kernels are scheduled onto the opposite engines.

Our Goal. As we illustrated, while an application-level offloading strategy can improve performance and energy efficiency, a finer, kernel-level offloading strategy can provide more opportunity. Therefore, our goal is to develop mechanisms for (1) automatically identifying the architectural affinity (GPU-PIM or GPU-PIC) of each kernel in an application, and (2) scheduling kernels that can concurrently execute on different parts of the PIM-assisted GPU architecture (GPU-PIM or GPU-PIC), while balancing the execution times across architectures and maintaining a kernel's architecture affinity as much as possible.

4. KERNEL OFFLOADING MECHANISM

This section presents an architecture affinity prediction model to enable a runtime mechanism for kernel offloading in PIM-Assisted GPU architectures.

Need for a Prediction Model. If *all* the kernels of an application prefer the same compute engine (either GPU-PIM or GPU-PIC), kernel offloading becomes equivalent to application offloading. However, if different kernels have different affinities, kernel-level offloading can yield higher performance than application offloading (Section 3.2). Because the CPU offloads computation to the GPU at the kernel granularity, maintaining the same granularity for PIM execution enables low-overhead computation offloading from the CPU to either GPU-PIC and GPU-PIM. To avoid the overhead of first sampling the performance of each kernel on both platforms and then deciding the appropriate platform for execution, we would like to *predict* the affinity of the kernel *before* it starts execution. To this end, we make use of a *regression model* that is composed of *predictive variables*.

Metrics. To build our regression model, we need to identify appropriate metrics (i.e., predictive variables) that can characterize the kernel affinity to GPU-PIC or GPU-PIM. We classify kernel characteristics into three primary categories: 1) memory intensity, 2) parallelism, and 3) shared memory (i.e., scratchpad) intensity. Table 1 lists these categories along with their predictive metrics/variables.

Table 1: Metrics used to predict compute engine affinity and GPU-PIC and GPU-PIM execution time.

Primary Category	Predictive Metric	Static/Dynamic
I: Memory intensity of Kernel	Memory to Compute Ratio	Static
	Number of Compute Inst.	Static
	Number of Memory Inst.	Static
II: Available Parallelism in the Kernel	Number of CTAs	Dynamic
	Total Number of Threads	Dynamic
	Number of Thread Inst.	Dynamic
III: Shared Memory Intensity of Kernel	Total Number of Shared Memory Inst.	Static

To measure the effect of *kernel memory intensity* on performance, we consider the memory-to-compute-ratio of the instruction mix executed by that particular kernel. Memory-to-compute ratio gives insight into the level of performance the kernel can achieve with higher memory bandwidth, as the computation requires data from the memory, indicating whether the higher bandwidth available on GPU-PIM is beneficial for performance. We use the *number of cooperative thread arrays (CTAs)* (also called work groups or thread blocks) as a measure of the parallelism in the kernel. This allows us to take into account the difference in the peak instruction throughput between GPU-PIC and GPU-PIM for each kernel. For a kernel with a high number of CTAs, GPU-PIC's performance might be higher than that of GPU-PIM due to the higher number of cores in GPU-PIC. To approximate a kernel's *shared memory intensity*, we measure the total number of shared memory instructions in the kernel. A shared-memory-intensive application might not require high DRAM bandwidth, making it more suitable for GPU-PIC. Predictive metrics/variables (listed in Table 1 middle column) are used to help build a robust model by complementing the primary metrics of the kernel. For example, in applications such as `RED, CONV, STRM`, the number of CTAs along with the number of threads provides a notion about the CTA sizes. Usually a larger CTA has higher resource requirements, which might lead to fewer of such CTAs to be scheduled onto an SM at any given time, even in the presence of available SM resources (but *not enough resources* to schedule another complete CTA). Therefore, having more SMs, as in GPU-PIC, might lead to better performance for such a large CTA. We classify the metrics as static or dynamic, as shown in Table 1. Static metrics are obtained by simply parsing the source code, while dynamic metrics are input-set-based and can be known only at/after kernel launch or during runtime.

Why These Metrics? We choose only the most influential metrics to build our regression model, i.e., those metrics that contribute the most to the model's accuracy. We define accuracy for our regression model for a kernel as either 100%, if the model predicts the kernel's affinity correctly, or 0%, if it predicts incorrectly. We experimented with building the regression model using the number of thread instructions as the only metric, and found that this model has an accuracy of 79% on the *training kernels*. Using all the metrics shown in Table 1 leads to an overall accuracy of 87%. The use of other characterization metrics described in Goswami et al. [34] does not further improve accuracy.

Table 2 provides the detailed characteristics of various application kernels, obtained via offline profiling. It also shows the primary category metrics (memory-to-compute-ratio, the number of CTAs, and shared memory intensity) and architecture affinity of each kernel (GPU-PIM (Y) or GPU-PIC (N)). The affinity of a kernel can be reasoned for most of the kernels by understanding the level of each metric category. For example, kernels such as `convolutionRows` and `prescan` have high shared memory intensity, and prefer to run on GPU-PIC because of the lower DRAM bandwidth demand and the higher instruction throughput available in GPU-PIC compared to GPU-PIM. Kernels of `GRAM` prefer GPU-PIM because of their lower parallelism and high memory intensity. Kernels such as `fdtd_step1_kernel`, which have high memory intensity coupled with high parallelism, prefer GPU-PIM, as they benefit from the higher available memory bandwidth.

Regression Model for Affinity Prediction. We build a logistic regression model [30, 43], shown in Equation 1, which provides a prediction for the affinity of a given kernel. A logistic regression model is a classifier and it generates a discrete output $\sigma(t)$: **1 for GPU-PIM** and **0 for GPU-PIC**. The model uses the metrics in Table 1 as inputs.

$$\sigma(t) = \frac{e^t}{e^t + 1} \quad (1)$$

where:

$$\begin{aligned} \sigma(t) &= \text{model output } (\sigma(t) < 0.5 \Rightarrow 0,\ \sigma(t) \geq 0.5 \Rightarrow 1) \\ t &= \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4 + \alpha_5 x_5 + \alpha_6 x_6 + \alpha_7 x_7 \\ \alpha_i &= \text{Coefficients of the Regression Model} \\ x_i &= \text{Predictive Metrics/Variables (Table 1)} \end{aligned}$$

To train the logistic regression model, we randomly sample 60% (15) of the 25 GPGPU applications considered in the paper. These 15 applications consist of 82 unique kernels that are used as inputs for the *training* of the model. The remaining 40% (10) of the applications, consisting of 42 unique kernels, are part of the *testing* set and are used to validate the model. We perform offline profiling of entire application execution only once to train the regression model and use this built model at runtime for affinity prediction. The model is able to accurately predict the architecture affinity (either GPU-PIC or GPU-PIM) of 83% of the test kernels. Table 2 shows the affinity prediction (in column D) of our model for each kernel along with the true affinity of the kernel (in column B). The major sources of inaccuracy are due to cache effects and branch divergence, which our regression model fails to accurately capture because of their heavily runtime-dependent characteristics. For example, due to the considerable amount of cache hits in kernels such as `drelax (BFS)` and `dfindelemin (MST)`, GPU-PIC outperforms GPU-PIM, even though these kernels have high parallelism and memory intensity. Yet, our model estimates incorrectly that they are better executed on GPU-PIM. We further discuss the effects of random sampling and application input on our model's accuracy in Section 8.

Kernel affinity could also be predicted statically, without using the dynamic metrics in Table 1. Using only the static metrics, the accuracy of the affinity prediction model decreases from 83% to 74%. We find that the inclusion of dynamic (i.e., input-based) metrics in the model is necessary for accurate prediction in applications such as CFD, STRM and PVC. This is because the affinity of these applications' kernels is highly influenced by input-based kernel dimensions (i.e., number of threads, CTA size) as they significantly affect the compute/resource requirements of these kernels.

Implementation. Figure 7 shows our framework to enable kernel offloading to PIM engines. Before runtime, a simple source to source translation is performed ❶. The purpose is to compute the values of the static metrics, such as the number of memory/compute/shared-memory instructions (in terms of PTX instructions), and embed them as arguments to the kernel launch call. We extend the CUDA runtime to implement the architecture affinity prediction model ❶❶. The prediction model is trained *offline* and does not incur any overhead of training during online prediction of affinity. At runtime, during kernel launch from the CPU, the dynamic metrics required for the prediction model such as the number of CTAs and the number of threads get populated with their values in the kernel launch call. Using both the static and dynamic metrics, the CUDA runtime on the

Table 2: Kernel characteristics, classification, and architecture affinity. Legend: (I) Memory Intensity (Memory to Compute Ratio = $L :\leq 0.2, 0.2 < M \leq 0.3, H :> 0.3$), (II) Parallelism (No. of CTAs = $L :\leq 64, 64 < M \leq 1024, H :> 1024$), (III) Shared Memory Intensity (Total no. of Shared Mem. Inst. = $L :\leq 2.5 \times 10^5, H :> 2.5 \times 10^5$). (B) Architecture affinity (Y: GPU-PIM, N: GPU-PIC), (C) Major reasons for architecture affinity, (D) Affinity prediction by our regression model in Section 4) (Y: GPU-PIM, N: GPU-PIC). Only kernels that dominate application execution time are shown.

Workload	Kernel Name	I	II	III	B	C	D	Workload	Kernel Name	I	II	III	B	C	D
BFS [17]	initialize	M	H	L	Y	Cache/BW	Y	RED [25]	reduce	H	M	L	Y	BW	N
	drelax	H	H	L	N	Cache	N	SCP [81]	scalarProdGPU	L	M	L	Y	—	N
BICG [37]	bicg_kernel1	H	L	L	Y	BW	Y	SLA [81]	prescan	L	H	H	N	S.Mem.	N
	bicg_kernel2	M	L	L	N	Cache	Y	STRM [21]	kernel_compute_cost	H	M	L	Y	BW	Y
BLK [81]	BlackScholesGPU	L	M	L	Y	Cache	Y		sgemmNN_MinPlus	L	L	H	Y	Lat.	Y
CCL [42]	MapperCount	H	M	L	Y	BW	Y	APSP [16]	matrixMul	H	M	L	Y	Lat.	N
	unitBitonicSortKernel	L	H	M	N	Compute	N		apsp_seq	M	L	L	Y	Lat.	Y
	prescan	M	M	H	N	S.Mem.	Y		cuda_compute_step_factor	M	M	L	Y	BW	Y
CONV [37]	convolution3D_kernel	M	M	L	N	Cache	Y	CFD [21]	cuda_compute_flux	L	M	L	N	Cache	Y
CORR [37]	std_kernel	M	L	L	Y	BW	Y		cuda_time_step	H	L	L	Y	BW	Y
	reduce_kernel	M	H	L	N	Compute	N	CONS [81]	convolutionRows	H	H	H	N	S.Mem.	N
	corr_kernel	L	L	L	N	Cache	Y		convolutionColumns	H	H	H	N	S.Mem.	N
FDTD [37]	fdtd_step1_kernel	H	H	L	Y	BW	Y	FWT [81]	fwtBatch1Kernel	H	H	L	Y	BW	Y
	fdtd_step2_kernel	M	H	L	N	Cache	N	GUPS	RandomAccessUpdate	L	M	L	Y	BW	Y
	fdtd_step3_kernel	H	H	L	N	Cache	N	LIB [81]	Pathcalc_Portfolio_KernelGPU	L	L	L	N	Cache	N
GRAM [37]	gramschmidt_kernel1	M	L	L	Y	Lat.	Y		Pathcalc_Portfolio_KernelGPU2	L	L	L	Y	BW	Y
	gramschmidt_kernel2	H	L	L	Y	Lat.	Y	MST [17]	dfindelemin	H	H	L	Y	Cache	Y
	gramschmidt_kernel3	M	L	L	Y	Lat.	Y		dfindcompmin	H	H	L	N	Cache	N
LUH [56]	IntegrateStress-ForElems_kernel	L	M	L	Y	BW/Lat.	Y		init	H	H	L	Y	Cache	Y
	CalcHourglassControl-ForElems_kernel	H	M	H	N	S.Mem.	N	PVC [42]	MapperCount	H	H	L	Y	BW	N
	CalcFBHourglassForce-ForElems_kernel	H	M	H	N	S.Mem.	N		prescan	L	H	H	N	S.Mem.	N
MVT [37]	mvt_kernel1	M	L	L	N	Cache	Y	SP [17]	dinit	H	H	L	Y	Cache	Y
	mvt_kernel2	H	L	L	Y	BW	Y		dupdateeta	H	H	L	Y	Cache	N
PR [42]	MapperCount	H	H	L	Y	BW	Y	TRA [81]	transpose_naive	M	H	L	Y	Cache	Y
	prescan	L	H	H	N	S.Mem.	N		transpose	L	H	L	Y	Cache	N

host-side computes[7] the architecture affinity of the kernel using the affinity prediction model, and offloads the kernel to the architecture that is expected to provide the highest performance. Doing so avoids the overhead of kernel migration, which might arise if the kernel were to be offloaded *after* it starts execution on a less preferred architecture.

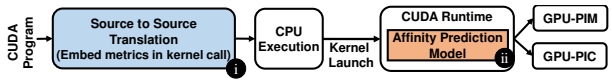


Figure 7: Modified CUDA runtime for kernel offloading

5. CONCURRENT KERNEL MANAGEMENT

This section presents a new runtime kernel management scheme to efficiently execute multiple kernels concurrently in the heterogeneous computational units in a PIM-assisted GPU architecture.

5.1 Analysis

Offloading kernels appropriately to their preferred compute engine (Section 4) leads to performance benefit. However, due to the sequential execution of the kernels, such PIM-assisted GPU architectures are under-utilized as only either the GPU-PIC or the GPU-PIM executes a kernel at a given time, but not both. If there are independent kernels, their concurrent execution on GPU-PIC and GPU-PIM can improve overall system utilization and performance (Section 3.2). We develop a mechanism to achieve such concurrent execution of kernels on both GPU-PIC and GPU-PIM. To efficiently schedule kernels for concurrent execution, we need three key pieces of information: (1) kernel-level dependence information, to identify independent kernels; (2) affinity of each kernel and (3) execution time prediction for each kernel, both to decide what compute engine is the best to execute the kernel on. We first describe how this information is gathered in our runtime system.

(I) Kernel-level Dependence Information. A kernel-level data dependence graph is required to decide which kernels can execute in parallel. For our evaluations, we obtain the dependence graph of an application *for a given input* by profiling the application's execution to determine the correct and complete set of read-after-write (RAW) dependencies across the kernels. Such cross-kernel dependencies can be easily found and marked by a compiler. Most compilers targeting array-based applications already run dependence analysis for each loop nest (kernel). One could extend such an analysis to check dependence's *across* loop nests (kernels) as well.[8]

Note that, our concurrent kernel management mechanism can be directly used for applications that inherently possess concurrent kernels (which could be conveyed by the programmer), without any need for data dependence analysis.

(II) Architecture Affinity Information. We observed in Section 3.2 that kernel affinity information can help improve performance of concurrent kernel execution (Scenario-V, Figure 6). To predict the affinity of a kernel, we use the logistic regression model described in Section 4. This model is used to fill the GPU-PIM and GPU-PIC queues with kernels based on their affinity.

(III) Execution Time Information. Kernel dependence and affinity information is necessary, but not sufficient to balance kernel execution times across GPU-PIC and GPU-PIM. For instance, consider an application consisting of two independent kernels having affinity towards GPU-PIM. If only the affinity information is used, both these kernels are offloaded to GPU-PIM, which leads to the under-

[7]The regression model parameters are kept in CPU memory. During the API call, model-based affinity is predicted by the CPU, which requires only 15 32-bit floating point operations (7 multiplications, 7 additions and 1 comparison).

[8]Note that not all applications have multiple independent kernels and thus can take advantage of concurrent kernel execution. We observe this in `SCP` and `GUPS`.

utilization of GPU-PIC (and perhaps a lost opportunity to improve performance). We address such situations by executing kernels in compute engines that do *not* satisfy the kernel affinity, if doing so would reduce overall execution time. Therefore, in this example, we might offload the kernel that has the lower execution time on GPU-PIC, to GPU-PIC. However, this requires the estimation of the kernel execution times on both GPU-PIC and GPU-PIM, for which we develop a model next.

5.2 Execution Time Prediction Model

For predicting a kernel's execution time, we build a linear regression model [74] that uses the same predictive metrics used in our architecture affinity prediction model (Table 1). Equation 2 shows the linear regression model.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6 + \beta_7 x_7 \quad (2)$$

where:

y = model output (predicted execution time; classified into bins using Table 3)
β_i = Coefficients of the Regression Model
x_i = Predictive Metrics/Variables (Table 1)

The model is trained using the kernel execution time information obtained from profiling the execution times of applications from the training set that were used in Section 4 to create the affinity prediction model. Earlier work (Dubach et al. [28]) utilizes compile-time parameters to predict the performance of an application, but exactly predicting the absolute execution time of a kernel accurately is a difficult task and exact prediction has significant error. Therefore, rather than utilizing the predicted execution time directly in an absolute manner, we classify the predicted execution time (y) into five different bins, as shown in Table 3. The ranges of the bins were carefully chosen by analyzing the profiled data. With such classification, we can efficiently schedule the kernels on GPU-PIC and GPU-PIM, without having to accurately predict the absolute execution times, albeit perhaps with lower benefit than if we had the correct absolute execution times.

Table 3: Classification of predicted execution time into bins.

Classification Bins	Very Low (1)	Low (2)	Medium (3)	High (4)	Very High (5)
Range (in Cycles)	<10K	10K-500K	500K-5M	5M-50M	>50M

Figure 8 shows the classification error of our regression-based execution time prediction models for GPU-PIC and GPU-PIM. Error is measured by calculating the distance of the predicted bin from the true bin normalized to the total number of bins. For example, a predicted bin of *Low (2)* with a true bin of *Very Low (1)* has an error of $(2-1)/5 = 0.2$, since the total number of bins is 5 and the distance of the predicted bin from the true bin is 1. The results show that the execution time prediction model provides a classification accuracy of 77% and 80% on the test set for GPU-PIC and GPU-PIM, respectively. The inaccuracies are mainly due to the heavily-runtime-dependent cache and branch divergence effects (as discussed in Section 4).

5.3 Algorithmic Details and Implementation

The main incentive to concurrently execute kernels is to maximize system utilization. For this purpose, the execution time on GPU-PIC and GPU-PIM needs to be balanced. However, the problem of balancing execution times across two architectures is equivalent to a known NP-Complete partitioning problem [32]. The partitioning problem is the task

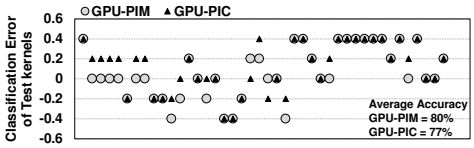


Figure 8: Classification error of test kernel execution times.

of deciding whether a given set of positive integers can be partitioned into two subsets such that the sum of the two subsets are equal. The only difference in our case is that we have a list of pairs (instead of a single number) of positive execution times, where each pair is a tuple of <execution time on GPU-PIC, execution time on GPU-PIM>. We adopt a greedy approach to solve this problem in three steps.

Figure 9 shows the schematic of our concurrent kernel management framework. The CUDA runtime predicts the architecture affinity (ii) of all kernels launched by the CPU. The kernels are fed into their respective queues in the kernel distribution unit (iv) and the execution begins. During runtime, there might be a case where all the kernels prefer a single type of compute engine or either the GPU-PIC or GPU-PIM queue is empty but there are still kernels waiting in the other compute engine's queue. To address this, the runtime steals a waiting independent kernel from the non-empty queue to the empty queue based on the kernel's execution time prediction (iii) on the less preferred architecture. The runtime prefers to steal the *first* independent kernel that has a predicted execution time (on the less preferred architecture) smaller than the remaining execution time of the currently executing kernel. Algorithm 1 describes this greedy process of kernel stealing that enables concurrent kernel management.

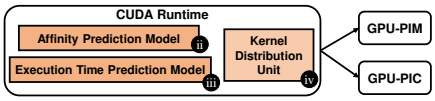


Figure 9: Modified CUDA runtime for concurrent kernel management.

Algorithm 1 Runtime Queue Management

▷ $time(kernel, engine)$ returns the time-bin of the kernel if it is executed on the given engine.
▷ Let X and Y represent the two different engines (GPU-PIC and GPU-PIM).
▷ $independent\_kernels(queue)$ returns the list of the kernels that can run concurrently with the kernels currently executing and have no past dependencies.
▷ $time\_executed(kernel)$ returns the amount of time the kernel has been executing.
if $X = Idle$ && $X.queue = \emptyset$ && $Y.queue \neq \emptyset$ **then**
  if $independent\_kernels(Y.queue) \neq \emptyset$ **then**
    for each kernel in $independent\_kernels(Y.queue)$ **do**
      if $time(kernel, X) \leq (time(kernel_{running,y}, Y) - time\_executed(kernel_{running,y}) + time(kernel, Y))$ **then**
        Execute "kernel" on X
        Break
▷ Similarly, repeat the process for Y

6. EVALUATION METHODOLOGY

Infrastructure. We modified the cycle-accurate GPGPU-Sim v3.2.2 [12, 52] to simulate our PIM-Assisted GPU architecture. To enable the execution of two different GPUs, we created two clusters of SMs, one for GPU-PIC and another for GPU-PIM. GPU-PIC and GPU-PIM do *not* concurrently work on the same data. Therefore, to maintain coherence between them, we flush the L2 cache in GPU-PIC after kernel execution. As the L1 caches are write-through,

they do not need to be flushed. There is no explicit synchronization needed between GPU-PIC and GPU-PIM as they never share data during concurrent execution. The kernel distribution unit in the CUDA runtime, shown in Figure 9, checks for any inter-kernel dependencies to avoid concurrent scheduling of dependent kernels.

For concurrent kernel execution, we use CUDA Streams, which are supported by GPGPU-Sim. A stream is a series of kernel launches and memory operations between the CPU and GPU that are executed sequentially. Different streams are capable of executing concurrently depending on available resources. In our simulation framework, we create two CUDA Streams: `PIC_CUDAStream` and `PIM_CUDAStream`. GPU-PIM and GPU-PIC are assigned their own streams, thereby making the execution of the streams concurrent. We modified the CUDA runtime support in GPGPU-Sim to overload the API calls with the kernel metrics and added support for the *architecture affinity prediction model* (Section 4), *execution time prediction model* (Section 5.2), and *kernel distribution unit* (Section 5.3). During application execution, our runtime framework is able to read the kernel metrics passed along with the kernel launch call, obtain the runtime metrics, and predict the affinity of the kernel.

Workloads. We chose CUDA applications from various benchmark suites (NVIDIA SDK [81], Rodinia [21], Shoc [25], Polybench [37], Mars [42], LonestarGPU [17]) and several other applications (e.g., LUH [56], APSP [16], GUPS). We collect the results at kernel boundaries to ensure that all comparisons are for the same amount of work completed across different executions. We only simulate the portions of code that are executed on GPU.

Simulated Systems. Table 4 provides the details of the simulated GPU-PIC and GPU-PIM configurations. We assume that GPU-PIC has 32 GPU cores and GPU-PIM has 8 GPU cores underneath a 3D memory stack. We compare this design to a conventional baseline GPU architecture that has 40 GPU cores. These two configurations have the same peak execution throughput. The L2 cache size of the baseline GPU is equal to that of GPU-PIC (768 kB). GPU-PIM does not have an L2 cache Section 2.2). All our simulations on the baseline GPU maintain (if any) inter-kernel data reuse, i.e., the L2 cache is *not* flushed in-between the execution of two kernels, unlike concurrent GPU-PIC and GPU-PIM execution on our proposed architecture.[9]

To simulate the timing of 3D-stacked DRAM, we use the timing parameters provided by Jevdjic et al. [49]. We use GPUWattch [65] for power analysis of GPU-PIC and GPU-PIM cores, caches, and interconnect. For DRAM energy analysis, we augment this model with a simple linear equation adding the wire transfer energy numbers given by Keckler et al. [58] to the DRAM read/write energy for a Hybrid Memory Cube DRAM [86]. We faithfully model the bandwidth, latency, and timing of 3D-stacked DRAM.

[9]We also considered using a baseline architecture capable of concurrent kernel execution on a *partitioned* set of 32 SMs and 8 SMs, but there are two issues with such a baseline. First, GPGPU applications tend to be highly parallel and fill the entire set of SMs provided, and we take this into account in our 40-SM baseline. Second, we still need a mechanism to decide which partition to launch each of the kernels. Depending on its characteristics, a kernel might prefer lower or higher number of SMs. Our new kernel offloading mechanism may be modified to make such a baseline work, but we leave this as future work. For these reasons, we normalize all results to a baseline with 40 SMs and no concurrent kernel execution.

Table 4: Parameters of the simulated system.

GPU-PIC Features	1.4GHz, 32 cores, 32 SIMT width, GTO warp scheduler [82]
Shared L2 (GPU-PIC)	16-way 64KB/memory channel, 128B cache block size
GPU-PIM Features	1.4GHz, 8 cores, 32 SIMT width, GTO warp scheduler [82]
Resources per Core [36, 91, 92]	16KB shared memory, 16KB register file, Max. 1536 threads (48 warps, 32 threads/warp)
Private Caches per Core [36, 91, 92]	16KB L1 D-cache, 12KB T-cache, 8KB C-cache, 2KB I-cache, 128B block size
Memory Model	12 Memory Controllers, FR-FCFS, 8 banks/MC, 924 MHz Partition chunk size: 256 bytes [35] t_{CL} = 11, t_{RP} = 11, t_{RC} = 39, t_{RAS} = 28, t_{CCD} = 2 t_{RCD} = 11, t_{RRD} = 5, t_{CDLR} = 5, t_{WR} = 12
Bandwidth	88.8GB/s (Interposer link per GPU-PIM stack) 355.2GB/s (within GPU-PIM stack)
Interconnect (GPU-PIC) [36]	1 crossbar/direction (32 cores, 12 MCs), flit size=32B, 1.4GHz, islip VC & switch allocators
Interconnect (GPU-PIM) [36]	1 crossbar/direction (8 cores, 12 MCs), flit size=32B, 1.4GHz, islip VC & switch allocators

GPU-PIM has access to the full bandwidth of 3D-stacked DRAM, whereas GPU-PIC has limited bandwidth as it is pin-limited. Table 5 provides the parameters we use for calculating DRAM energy. We assume a crossbar interconnect between the private L1 caches and the shared L2 cache, keeping it consistent with the currently available GPUs [6].

Table 5: Parameters of our DRAM energy model.

Energy per bit [86]	13.7 pJ/bit
Wire Energy (256 bits, 10 mm) [58]	310 pJ
Assumed distance between GPU-PIC and GPU-PIM	20 mm

7. EXPERIMENTAL RESULTS

We compare the performance and energy efficiency of our kernel offloading and concurrent kernel management schemes with their oracle counterparts. The oracle schemes make ideal offloading and concurrent execution decisions by profiling each kernel with its runtime input and obtaining completely accurate execution times for each kernel on GPU-PIC and GPU-PIM (instead of using regression models). Therefore, the oracle schemes provide the best possible performance for each application. The oracle kernel offloading scheme finds the correct kernel affinity for each kernel and schedules it for execution on the best engine. The oracle concurrent kernel management scheme provides the minimal execution time for an application by utilizing both GPU-PIC and GPU-PIM while respecting kernel dependencies. We *normalize* the performance and efficiency results to those of the 40-SM baseline GPU described in Section 6. We report the results separately for the applications used for training and testing, to show the effectiveness of our model.

Effects of Kernel Offloading. Figures 10a and 10b show the performance and energy efficiency benefits of our dynamic kernel offloading scheme, respectively. Our technique increases the performance and energy efficiency of the testing set applications by 25% and 28%, respectively. The oracle kernel offloading scheme provides 34% and 40% average performance and energy efficiency improvement, respectively, on the testing set. The inability of our scheme to perform as good as the oracle scheme is due to the mispredictions of kernel affinity by our regression model.[10]

The inaccuracies of our prediction model (Section 4) usually cause some kernels (e.g., of `CONV` and `CORR`) to be incorrectly offloaded to GPU-PIM, leading to performance losses.

[10]In some cases where computation parallelism is the main bottleneck, e.g., for `CONV`, even the oracle scheme loses performance compared to the baseline because the baseline can utilize all the 40 GPU cores for each kernel whereas the PIM-assisted GPU architecture can utilize either 32 or 8.

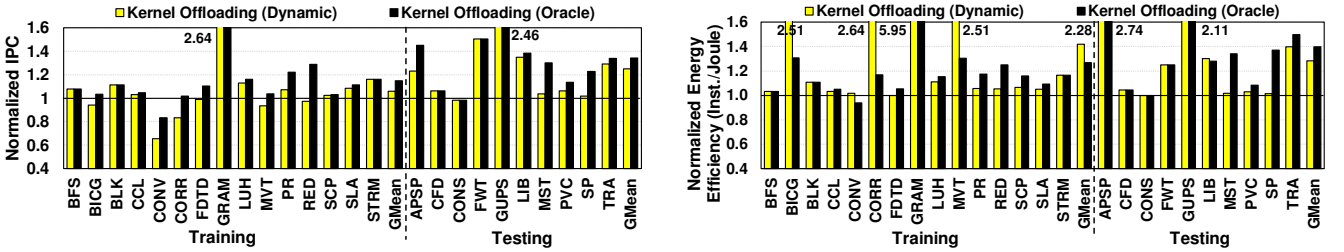


Figure 10: Impact of our Kernel Offloading scheme on (a) Performance, (b) Energy efficiency.

However, because GPU-PIM is significantly more energy efficient than GPU-PIC, in most of the mispredicted kernels (e.g., of APSP and LIB; see Figure 10b), the overall energy efficiency of our scheme is still higher than that of the oracle's, which is optimized for performance, not energy.

Figure 11 shows the percentage of execution time when GPU-PIC and GPU-PIM are executing kernels. Applications such as LUH, PR, and SLA have kernels that exhibit different architecture affinities within the application and thus benefit from kernel offloading over application offloading. Our model is able to capture this variation in affinity and we utilize the opportunity that kernel-level offloading presents us as mentioned in Section 4. In this scheme, GPU-PIC and GPU-PIM *do not execute concurrently*, leading to under-utilization of the system.

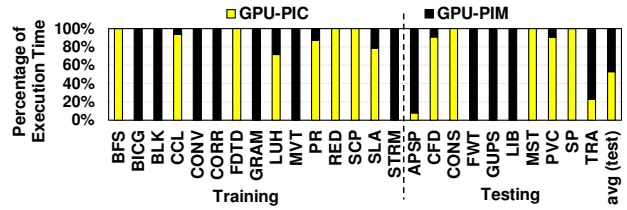


Figure 11: Percentage of execution time GPU-PIM and GPU-PIC execute kernels with our kernel offloading scheme.

We make three observations based on these results. First, kernel offloading gives the same results as the best application offloading (as we showed in Figure 4) for applications such as BFS, BLK, SCP, and GUPS. This is because these applications either consist of a single kernel (BLK, SCP and GUPS), or all the kernels of the application prefer the same engine (BFS). Second, kernel offloading performs better than application offloading in applications such as FDTD, PR, and PVC. This is due to the benefits of fine-grained offloading, discussed in Section 4, which selects the fastest execution engine for each kernel. Third, our scheme performs worse than application offloading for applications such as BICG, CORR, and MST, due to mispredictions in kernel affinity. BICG has two kernels, one of which prefers GPU-PIC and the other GPU-PIM. Due to mispredictions, the kernel that prefers execution on GPU-PIC is executed on GPU-PIM, leading to lower performance compared to application offloading, which offloads the entire application to GPU-PIC. These mispredictions are due to data reuse at the L2 cache, which make GPU-PIC faster than GPU-PIM (lacks an L2 cache).

Effects of Concurrent Kernel Management. Figures 12a and 12b show the performance and energy efficiency benefits of our concurrent kernel management scheme, respectively. On average, our management scheme improves performance and energy efficiency by 42±4% and 27±2%, respectively, over the baseline across the test set.[11] The oracle scheme on the test set provides 53% and 33% improvements in performance and energy efficiency, respectively. Similar to our kernel offloading scheme, affinity mispredictions cause performance penalties, but they improve energy efficiency in general.

Figure 13 shows the percentage of execution time when both GPU-PIC and GPU-PIM are executing kernels concurrently.[12] It can be seen that the applications with high concurrency across the engines are the ones that gain the most in terms of performance.

We make four observations based on these results. First, our scheme performs the same as application offloading in applications such as BLK, GRAM, FWT, and GUPS. There is no kernel-level concurrency in these applications (see Figure 13). BLK and GUPS consist of a single kernel; GRAM and FWT have no independent kernels, leading to sequential execution of kernels in all of these applications.

Second, applications such as FDTD, PR, LIB and MST fail to perform as good as the oracle scheme because their execution times are not predicted accurately (even though their

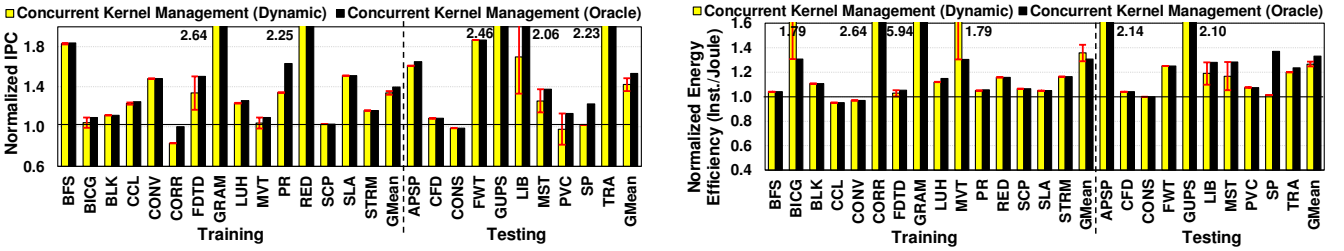


Figure 12: Impact of our Concurrent Kernel Management scheme on (a) Performance, (b) Energy efficiency.

[11] As a result of the dynamic nature of kernel scheduling and the greediness in our scheme, different concurrent kernel executions might be possible for the same application, which might result in different performance and efficiency. This arises because multiple independent kernels can be predicted to be in the *same execution time bin*. Any one of these kernels could be picked by our concurrent kernel scheduler as they are at equal "priority". The error bounds in Figure 12a and 12b show the variation in performance and energy efficiency due to different choices made by the scheduler at runtime.

[12] The breakdown for the best-case of concurrent kernel execution is shown.

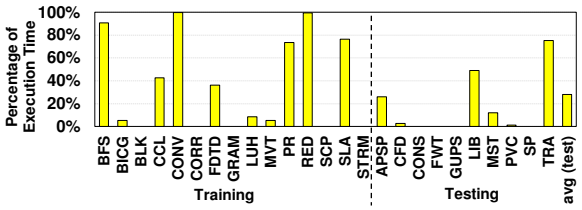


Figure 13: Percentage of execution time when kernels are concurrently running on GPU-PIM and GPU-PIC with our concurrent kernel management scheme.

affinity predictions are correct). Our heuristic checks for the kernel's predicted runtime on the unsuitable platform, and decides whether to wait for its preferred platform to be free, or to execute it concurrently on the less-suitable yet under-utilized platform right away. Our execution time prediction fails to calculate the exact execution time bin, and leads to sub-optimal scheduling decisions in these applications. Third, applications such as `BICG`, `MVT`, `PVC` and `SP` under-perform due to the mispredictions in kernel affinity. In `SP`, one of the two kernels is mispredicted for GPU-PIM, and both of the kernels are executed on GPU-PIM. Following our heuristic of running the kernel with shorter execution time based on the execution time bins does not help because both kernels fall into the same bin. Therefore, the heuristic arbitrarily chooses one kernel for GPU-PIM and the other for GPU-PIC, which negatively affects performance as one of the kernels takes considerably longer on GPU-PIC even though they fall in the same execution time classification bin. Also, due to the lack of many kernels, there is no leeway for mispredictions. The scenario is similar for the other applications as well. `APSP` also suffers from this situation, but because this application has many (200+) kernels that execute, the mispredicted kernels do not dominate the execution time. Thus, we still get significant performance improvement on `APSP`. Finally, for applications such as `CCL`, `RED`, `CFD` and `FWT`, we are able to achieve comparable performance to that of the oracle scheme, which is very significant. In `RED`, even though the affinity is incorrectly predicted, due to kernel stealing, we are able to offload the appropriate kernels to their preferred architecture, which leads to significant performance improvement.

We conclude that our kernel offloading and concurrent kernel management schemes lead to significant average performance and energy efficiency improvements across 25 applications we evaluated for both training and testing.

8. SENSITIVITY STUDIES

We perform multiple sensitivity studies to understand: 1) the impact of architectural decisions for GPU-PIM, 2) sensitivity of the regression model to the testing set and different application inputs, 3) opportunities and challenges of utilizing multiple GPU-PIMs.

8.1 GPU-PIM Design Choices

L2 Cache and Core Count. We analyzed the effect of caches on GPU-PIM's performance. There is 9% slowdown when a 384kB L2 cache is added to GPU-PIM, due to the additional latency the L2 cache introduces to the memory request path, which is heavily used in the presence of memory-intensive kernels. We also varied the number of SMs in GPU-PIM from 4 to 16. With 8 SMs, GPU-PIM achieves within 30% of the performance of 16 SMs. Importantly, 8 SMs is thermally feasible (< 50W), but more SMs likely reduce the feasibility of GPU-PIM.

8.2 Regression Model

Training Set. To evaluate our mechanisms' sensitivity to applications used in the training set, we selected a completely different training set (by picking applications to be included in the training set in a random manner) and rebuilt the regression model. We found the accuracy in predicting the architecture affinity of the test kernels to be 81%. We also performed a semi-random sampling, where 20% of the *most influential* applications (that affect the accuracy of the model) were made a part of the training set and the remaining 40% were chosen randomly to be included in the training set. This leads to an accuracy of 88% for the test kernels. To study the impact of varying the size of the training set on the regression model, we build models using three different training set sizes: 40%, 60% and 80% of the applications. The accuracy of these models on the test set were 70%, 83% and 81%, respectively. As the training set size increases, accuracy improves to a point, but further increase over-fits the model (i.e., when more than 60% of the applications in the training set), after which accuracy starts decreasing.

Sensitivity to Application Input Set. Figure 14 shows the performance of our kernel offloading scheme on 8 GPGPU applications with different input sets. For applications such as `CFD, STRM, PVC`, we are able to predict the affinity accurately. These applications are able to change their parallelism (number of CTAs, number of threads) depending on their inputs, enabling accurate affinity prediction. For `RED Input-2`, we capture affinity accurately, but `RED Input-1` has mispredictions. This is because Input-1 fits inside the cache and has high reuse, unlike Input-2, whose working set is larger than the cache. For applications such as `BFS, MST, SP`, we predict affinity incorrectly, which leads to suboptimal performance. These applications are heavily dependent on their input data and it is difficult for our model to perfectly account for such very irregular cases. In `MST Input-1`, different kernels prefer different affinities, whereas `MST Input-2` contains kernels that always prefer to execute on GPU-PIM.

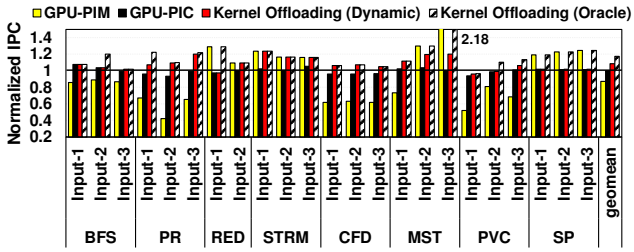


Figure 14: Affinity prediction model's sensitivity to input.

8.3 Systems with Multiple GPU-PIMs

Number of GPU-PIMs. We experimented with two other configurations that have 2 and 4 GPU-PIMs, respectively, while increasing the GPU-PIC memory bandwidth by 2× and 4× (due to multiple memory links between GPU-PIC and GPU-PIMs). Assuming perfect data placement (i.e., data needed by a GPU-PIM is located in its local stacked memory), application offloading onto only GPU-PIMs gives an improvement of 31% and 51% in performance and energy efficiency, respectively. With the *best application offloading* scheme, we see improvements of 53% and 65%, re-

spectively. These trends are similar to the trends discussed in Section 3, and indicate that our schemes are likely to be scalable with number of GPU-PIMs.

Data Placement. Earlier results with multiple GPU-PIMs assumed *perfect* data placement such that each GPU-PIM has the data it needs in its local memory stack. However, this might not always be possible. Therefore, architectures comprising multiple GPU-PIMs might suffer from memory transactions that are requested and served by different (non-local) memory stacks. Moreover, these transactions have to go through the GPU-PIC since there are no direct communication channels between the different GPU-PIMs. With the default CTA scheduler and the default address mapping, we find that approximately 50% of memory transactions are non-local. We changed the striping of data from 256Bytes/bank to 32KBytes/bank, causing non-local accesses to reduce by 8-10% for applications such as `MST`, `STRM` and `LIB`. By modifying the CTA scheduler and/or address mapping intelligently, it is possible to minimize non-local accesses, as shown by a recent work [44].

9. RELATED WORK

To our knowledge, this is the first work that comprehensively investigates kernel-level offloading mechanisms in PIM-Assisted GPU Architectures. It is the first to develop automated models and methods for (1) offloading kernels to PIM units, (2) concurrently executing kernels on different heterogeneous compute engines in a PIM-Assisted GPU Architecture, to maximize system performance and efficiency. We briefly discuss research in related areas.

Processing-in-Memory (PIM) Architectures. There is a substantial body of work on PIM that explores placing computation units within memory (e.g., [18, 33, 40, 55, 62, 67, 85, 89, 90, 93–96, 98, 99]). 3D-stacked memory technology brings new dimensions and better feasibility to PIM-based architectures [3, 4, 14, 15, 27, 39, 41, 44, 70, 71, 73, 87, 109]. Our work is most closely related to the concurrent work of Hsieh et al. [44], which proposes programmer-transparent schemes for offloading code segments to PIM cores and for co-locating code and data together in a 3D-memory stack. They use a compiler-based technique to find the code segments to offload to the PIM compute units based on a cost-benefit analysis. Our work does not require sophisticated compiler support as it performs scheduling at the kernel level and kernels are already well designated in modern GPU applications. Farmahini-Farahani et al. [31] propose an architecture that reduces data transfers by stacking accelerators on top of off-chip DRAM devices. In the context of GPUs, Zhang et al. [112] propose TOP-PIM, a throughput-oriented PIM-Assisted GPU architecture. They show significant energy efficiency improvements by offloading GPU applications closer to memory. However, they evaluate executing the *entire application* on either the host or GPU-PIM. Our work builds upon a similar architecture and proposes mechanisms to more efficiently utilize such an architecture by performing scheduling at the finer-grained kernel level.

Machine Learning-based Prediction Models. Machine learning techniques have been widely deployed for performance and power prediction models (e.g., [7, 11, 45–47, 66, 72, 80, 84, 110, 113]). Wu et al. [110] propose a GPU performance and power model that uses machine learning techniques to predict the behavior of incoming applications from profiled data. Ardalani et al. [7] propose a performance prediction model that takes as input a single-threaded CPU version of an application and predicts the performance for its GPU port. Panwar et al. [84] present an online kernel characterization technique and performance model to estimate the performance of a kernel on different GPU architectures. Ipek et al. [45,46] develop models for performance prediction of parallel applications and for aiding architectural space exploration. We use a regression-based approach to develop our affinity and execution time prediction models for PIM-Assisted GPU architectures.

Task Scheduling. There has been considerable work done in the domain of task scheduling to improve load balance and performance in both homogeneous and heterogeneous systems (e.g., [8,19,23,26,38,50,51,97,103–105,111]). Aji et al. [5] design an OpenCL runtime called MultiCL, which can effectively map the command queues onto the best device for high performance. Their runtime scheduler involves static device profiling, dynamic kernel profiling, and dynamic device mapping. Chen et al. [22] study a task queue based dynamic load balancing mechanism for a multi-GPU setup. None of these works examine scheduling or load balancing issues in a PIM-Assisted GPU Architecture. In this paper, we develop new affinity and execution time prediction models to efficiently schedule kernels to heterogeneous compute units in such an architecture.

10. CONCLUSION

We developed two new code scheduling techniques that enable effective use of processing-in-memory (PIM) mechanisms in PIM-Assisted GPU architectures, where a conventional GPU is augmented with 3D memory stacks that house simpler GPUs in their logic layers. First, a *kernel offloading* mechanism that accurately decides what code portions to offload to the GPUs in the 3D memory stack, using a new regression-based kernel affinity prediction model. Second, a *concurrent kernel management* mechanism that uses the affinity prediction model, a new kernel execution time prediction model, and kernel dependency information to decide which kernels to schedule concurrently on both the main GPU cores and the GPU cores in memory stacks. These two mechanisms operate at the *kernel-level*, which simplifies scheduling and management, and makes our approach transparent to programmers and compilers, as code is offloaded from the host system to a GPU system at the kernel granularity in modern systems. We have comprehensively evaluated both of our mechanisms and shown that 1) they provide significant performance and energy efficiency improvements across a wide variety of GPU applications and 2) the improvements we obtain are robust to changes in the training set for our regression model and changes in system parameters. We conclude that our kernel-level scheduling mechanisms can be an effective runtime solution for exploiting processing-in-memory in modern GPU-based architectures.

ACKNOWLEDGMENTS

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

REFERENCES

[1] "The Green500 List - June 2015."
[2] "Top500 Supercomputer Sites - June 2015."
[3] J. Ahn *et al.*, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[4] J. Ahn *et al.*, "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-In-Memory Architecture," in *ISCA*, 2015.
[5] A. M. Aji *et al.*, "Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL," in *CLUSTER*, 2015.
[6] AMD, "Graphics Cores Next (GCN) Architecture," 2012.
[7] N. Ardalani *et al.*, "Cross-Architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance," in *MICRO*, 2015.
[8] C. Augonnet *et al.*, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, 2011.
[9] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[10] R. Ausavarungnirun *et al.*, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *PACT*, 2015.
[11] P. E. Bailey *et al.*, "Adaptive Configuration Selection for Power-constrained Heterogeneous Systems," in *ICPP*, 2014.
[12] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[13] R. Balasubramonian *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," in *IEEE Micro*, 2014.
[14] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2016.
[15] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2014.
[16] A. Buluç *et al.*, "Solving Path Problems on the GPU," *Parallel Computing*, 2010.
[17] M. Burtscher *et al.*, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.
[18] J. Carter *et al.*, "Impulse: Building a Smarter Memory Controller," in *HPCA*, 1999.
[19] R. Chandra *et al.*, "Scheduling and Page Migration for Multiprocessor Compute Servers," in *ASPLOS*, 1994.
[20] K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[21] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.
[22] L. Chen *et al.*, "Dynamic Load Balancing on Single-and Multi-GPU Systems," in *IPDPS*, 2010.
[23] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *JPDC*, 1989.
[24] W. J. Dally, "Challenges for Future Computing Systems," *HiPEAC Keynote*, 2015.
[25] A. Danalis *et al.*, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *GPGPU*, 2010.
[26] R. Das *et al.*, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *HPCA*, 2013.
[27] J. Draper *et al.*, "The Architecture of the DIVA Processing-in-memory Chip," in *ICS*, 2002.
[28] C. Dubach *et al.*, "Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction," in *CF*, 2007.
[29] Y. Eckert *et al.*, "Thermal Feasibility of Die-Stacked Processing in Memory," in *WoNDP*, 2014.
[30] R.-E. Fan *et al.*, "LIBLINEAR: A Library for Large Linear Classification," in *JMLR*, 2008.
[31] A. Farmahini-Farahani *et al.*, "DRAMA: An Architecture for Accelerated Processing near Memory," *IEEE CAL*, 2014.
[32] M. R. Garey *et al.*, "Some Simplified NP-complete Problems," in *STOC*, 1974.
[33] M. Gokhale *et al.*, "Processing in Memory: the Terasys Massively Parallel PIM Array," *IEEE Computer*, 1995.
[34] N. Goswami *et al.*, "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications," in *IISWC*, 2010.
[35] GPGPU-Sim v3.2.1. Address mapping.
[36] GPGPU-Sim v3.2.1. GTX 480 Configuration.
[37] S. Grauer-Gray *et al.*, "Auto-tuning a High-level Language targeted to GPU Codes," in *InPar*, 2012.
[38] C. Gregg *et al.*, "Dynamic Heterogeneous Scheduling Decisions using Historical Runtime Data," in *A4MMC*, 2011.
[39] Q. Guo *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[40] M. Hall *et al.*, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *SC*, 1999.
[41] M. Hashemi *et al.*, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
[42] B. He *et al.*, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.
[43] D. W. Hosmer and S. Lemeshow, *Applied Logistic Regression*, 2000.
[44] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[45] E. Ipek *et al.*, "An Approach to Performance Prediction for Parallel Applications," in *Euro-Par*, 2005.
[46] E. Ipek *et al.*, "Efficiently exploring architectural design spaces via predictive modeling," in *ASPLOS*, 2006.
[47] E. Ipek *et al.*, "Self Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.
[48] JEDEC, *JESD235 High Bandwidth Memory (HBM) DRAM*, Oct. 2013.
[49] D. Jevdjic *et al.*, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *MICRO*, 2014.
[50] J. A. Joao *et al.*, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *ASPLOS*, 2012.
[51] J. A. Joao *et al.*, "Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *ISCA*, 2013.
[52] A. Jog *et al.*, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.
[53] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.
[54] A. Jog *et al.*, "Exploiting Core Criticality for Enhanced Performance in GPUs," in *SIGMETRICS*, 2016.
[55] Y. Kang *et al.*, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 1999.
[56] I. Karlin *et al.*, "Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application," in *IPDPS*, 2013.
[57] O. Kayiran *et al.*, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.
[58] S. Keckler *et al.*, "GPUs and the Future of Parallel Computing," in *IEEE Micro*, 2011.
[59] H. Kim *et al.*, "Bounding Memory Interference Delay in COTS-based Multi-Core Systems," in *RTAS*, 2014.
[60] Y. Kim *et al.*, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[61] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[62] P. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," in *ICPP*, 1994.

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

[64] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," in *ACM TACO*, 2016.

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

[66] T. Li and L. K. John, "Run-time Modeling and Estimation of Operating System Power Consumption," in *SIGMETRICS*, 2003.

[67] G. Lipovski and C. Yu, "The Dynamic Associative Access Memory Chip and Its Application to SIMD Processing and Full-Text Database Retrieval," in *MTDT*, 1999.

[68] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[69] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[70] G. H. Loh *et al.*, "A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM," in *WoNDP*, 2013.

[71] G. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *ISCA*, 2008.

[72] K. Ma *et al.*, "GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures," in *ICPP*, 2012.

[73] S. Marko *et al.*, "Processing-in-Memory: Exploring the Design Space," in *ARCS*, 2015.

[74] D. C. Montgomery and E. Peck, *Introduction to Linear Regression Analysis*, 1992.

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

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

[77] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.

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

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

[80] H. Nagasaka *et al.*, "Statistical Power Modeling of GPU Kernels using Performance Counters," in *IGCC*, 2010.

[81] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011.

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

[83] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Maxwell GM20x," 2015.

[84] L. S. Panwar *et al.*, "Online Performance Projection for Clusters with Heterogeneous GPUs," in *ICPADS*, 2013.

[85] D. Patterson *et al.*, "A Case for Intelligent RAM," in *IEEE Micro*, 1997.

[86] J. T. Pawlowski, "Hybrid Memory Cube," *Hotchips*, 2011.

[87] S. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.

[88] M. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.

[89] A. Rahimi *et al.*, "Energy-Efficient GPGPU Architectures via Collaborative Compilation and Memristive Memory-Based Computing," in *DAC*, 2014.

[90] S. F. Reddaway, "DAP - a Distributed Array Processor," in *ISCA*, 1973.

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

[92] T. G. Rogers *et al.*, "Divergence-Aware Warp Scheduling," in *MICRO*, 2013.

[93] G. E. Sayre, "STARAN: An Associative Approach to Multiprocessor Architecture," in *Computer Architecture, Workshop of the Gesellschaft fur Informatik*, 1976.

[94] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE CAL*, 2015.

[95] V. Seshadri *et al.*, "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[96] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," in *MICRO*, 2015.

[97] B. A. Shirazi *et al.*, *Scheduling and Load Balancing in Parallel and Distributed Systems*, 1995.

[98] D. Smitley and K. Iobst, "Bit-Serial SIMD on the CM-2 and the Cray 2," in *SIAM PP*, 1990.

[99] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.

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

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

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

[103] M. A. Suleman *et al.*, "Data marshaling for multi-core architectures," in *ISCA*, 2010.

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

[105] H. Topcuoglu *et al.*, "Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing," *IEEE TPDS*, 2002.

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

[107] N. Vijaykumar *et al.*, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.

[108] H. Wang *et al.*, "A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters," in *VEE*, 2015.

[109] D. H. Woo *et al.*, "An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth," in *HPCA*, 2010.

[110] G. Wu *et al.*, "GPGPU Performance and Power Estimation using Machine Learning," in *HPCA*, 2015.

[111] C. Xu and F. C. Lau, *Load Balancing in Parallel Computers: Theory and Practice*, 1996.

[112] D. Zhang *et al.*, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.

[113] L. Zhang *et al.*, "Accurate Online Power Estimation and Automatic Battery Behavior based Power Model Generation for Smartphones," in *CODES+ISSS*, 2010.

[114] J. Zhao and Y. Xie, "Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration," in *ICCAD*, 2012.