INTGRATING PROCESSING IN-MEMORY (PIM) TECHNOLOGY INTO GENERAL PURPOSE GRAPHICS PROCESSING UNITS (GPGPU) FOR ENERGY EFFICIENT COMPUTING

A Thesis

Presented to

the Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in Electrical Engineering

by

Paraag Ashok Kumar Megharaj

August 2017

INTGRATING PROCESSING IN-MEMORY (PIM) TECHNOLOGY INTO GENERAL PURPOSE GRAPHICS PROCESSING UNITS (GPGPU) FOR ENERGY EFFICIENT COMPUTING

Paraag Ashok Kumar Megharaj

Approved:

Chair of the Committee
Dr. Xin Fu, Assistant Professor,
Electrical and Computer Engineering

Committee Members:

Dr. Jinghong Chen,
Associate Professor,
Electrical and Computer Engineering

Dr. Xuqing Wu, Assistant Professor,
Information and Logistics
Technology

Dr. Suresh K. Khator, Associate Dean,
Cullen College of Engineering

Dr. Badri Roysam, Professor and
Chair of Dept. in Electrical and
Computer Engineering

Acknowledgements

I would like to thank my advisor, Dr. Xin Fu, for providing me an opportunity to study under her guidance and encouragement through my graduate study at University of Houston.

I would like to thank Dr. Jinghong Chen and Dr. Xuqing Wu for serving on my thesis committee.

Besides, I want to thank my friend, Chenhao Xie, for helping me out and discussing in detail all the problems during the research.

I would also like to thank my parents for their support and encouragement to pursue a master's degree.

Finally, I want to thank my girlfriend, Shalini Singh, for all her love and support.

INTEGRATING PROCESSING IN-MEMORY (PIM) TECHNOLOGY INTO GENERAL PURPOSE GRAPHICS PROCESSING UNIT (GPGPU) FOR ENERGY EFFICIENT COMPUTING

An Abstract

of a

Thesis

Presented to

the Faculty of the Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in Electrical Engineering

by

Paraag Ashok Kumar Megharaj

August 2017

Abstract

Processing-in-memory (PIM) offers a viable solution to overcome the memory wall crisis that has been plaguing memory system for decades. Due to advancements in 3D stacking technology in recent years, PIM provides an opportunity to reduce both energy and data movement overheads, which are the primary concerns in present computer architecture community. General purpose GPU (GPGPU) systems, with most of its emerging applications data intensive, require large volume of data to be transferred at fast pace to keep the computations in processing units running, thereby putting an enormous pressure on the memory systems.

To explore the potential of PIM technology in solving the memory wall problem, in this research, we integrate PIM technology with GPGPU systems and develop a mechanism that dynamically identifies and offloads candidate thread blocks to PIM cores. Our offloading mechanism shows significant performance improvement (30% by average and up to 2.1x) as compared to the baseline GPGPU system without block offloading.

Table of Contents

List of Figures

List of Tables

1 Introduction

In recent years, main memory systems have become well known to be the critical bottlenecks for performance in majority of modern day general purpose GPU applications. This is mainly because memory systems haven't been unable to keep pace with the rapid improvements in GPU processing cores. In conjunction with incredible capabilities in handling different kinds of complex computations at high instructions per cycle, processing cores have also developed good energy efficiency. These rapid developments can be credited to the advent of multi-core and multi-threaded system architectures, aggressive pipelines designs and various scheduling techniques. Improvement in bandwidth, latency and energy consumption of off-chip memory system have not kept in pace with these advances [7, 8]. Hence, the memory system often becomes a bottleneck and accounts for significant system level energy consumption [10].

There are two major problems that today's GPU systems encounter frequently:

- *Insufficient memory bandwidth to meet the demands of GPU multi-core processor chips* - This insufficiency gets adverse as the number of cores on the chip increases. This is the primary reason why memory bandwidth issue is very frequently (almost every time by programmers) experienced in GPU systems as their architecture commands thousands of cores, all running in parallel.
- *Power consumption* - As systems expand in capabilities and multiple systems collaborating with each other to speed up application, power consumption by these systems become equally important factors in deciding its reliability.

With the emergence of new 3D stacked memory technology, problem of bottlenecks due to insufficient memory bandwidth can be solved effectively [33]. This is possible due to the high memory bandwidth, low memory access latency and low energy consumption in this technology. However the most important feature of stacking technology is that it enables close coupling of processing logic and the memory, hence *processing in-memory*. This is a very major advantage of this technology making it very promising for optimizing a large range of applications [34, 35]. Coupling logic very close to stacked memory is important because it enables us to completely utilize high bandwidth from stacked memory which otherwise would be limited to the bandwidth offered by the off chip links.

To further validate effectiveness of this technology, we explore the industry advancements in this technology which is very evident with Samsung Electronics and SAP co-developing in-memory technology. Hybrid Memory Cubes (HMC) 2.0 introduced by Micron which exploits stacking technology and re-architects the DRAM banks to achieve better timing and energy efficiency at a much smaller area footprint [16, 17]. A JEDEC standard for high performance applications, High Bandwidth Memory (HBM) [12]. In addition a number of academic publications have also explored stacked DRAM on logic dies [13, 14, 15].

Hence we believe that enabling general purpose graphics processing units with processing in-memory technology can speed up performances and increase energy efficiency [1, 36]. Figure 1 shows a high level diagram of integrating processing in-memory (PIM) technology with GPGPU system. This system consists of host GPGPU

connected with stacks of DRAM memory and PIM logic cores using off chip serial links. There can be one or more number of processing cores with stacked memory.

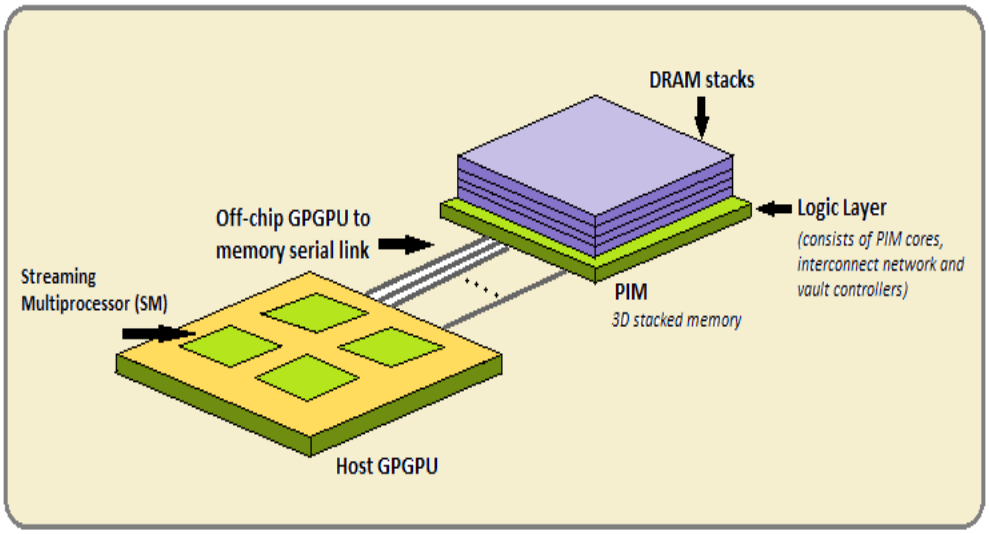

Figure 1: High level diagram of integration of *Processing in-Memory* (PIM) with *General Purpose Graphics Processing Unit* (GPGPU)

Offloading parts of execution workload from GPGPU to PIM can speed up performance and also ease bottlenecks due to memory access traffic on the off chip links.

In this research work, we want to explore and examine the potential of Processing-in-Memory (PIM) technology. Through this research work we intend to:

i. Model PIM enabled GPGPU and develop a technique that dynamically identifies and offloads parts of execution to a PIM cores.

ii. Effectively utilize the stacked memory resources i.e. high internal memory bandwidth, in a fast and energy efficient way.

iii. Evaluate the effectiveness of this technology with GPGPU system and investigate how PIM will impact the energy consumption on GPGPU workloads.

2. Motivation

The major challenge in working with PIM technology is deciding which part of the workload must be offloaded to the PIM cores. Most prior research in this area demonstrate that memory intensive instructions are the most favorable candidates for offload [1, 6, 37]. These candidates could be identified with help from the programmer who will specify, to best of their knowledge, which part of the code is memory intensive and should be run on PIM cores [21, 36, 38]. However, this is not a very good technique to identify candidates for offload, because here the programmer must be well versed with the underlying implementation and memory interaction. Also, with this technique, offloading becomes only as good as the programmer. Another technique to determine candidates for offload could be by combined efforts from the compiler and the underlying hardware [6, 37].

These methods only identify memory intensive parts of the code for offload. However, offloading parts of code (known as a kernel in GPU terminology) to the PIM does not always result in optimal performance benefit. When an instruction is being executed on GPU cores, different threads process the same instruction but on different data elements (data parallelism). Here we observe that even if a memory intensive instruction is being executed, different thread blocks (group of threads running on a GPU cores) have different memory intensities. Common examples of application constantly running on imbalanced workloads are graph computing applications [39]. An example of this is described in figure 2.

Figure 2, gives a distribution of thread blocks, *Cooperative Thread Arrays* (CTA) in CUDA terminology, being executed on a GPGPU core at different instances with their average execution times. In the figure, there are six blocks running on a core simultaneously at six different instances. We can observe that block execution time of different thread blocks can be highly imbalanced with fastest thread blocks executing at an average of 700 cycles and slowest ones executing in around 4000 cycles.

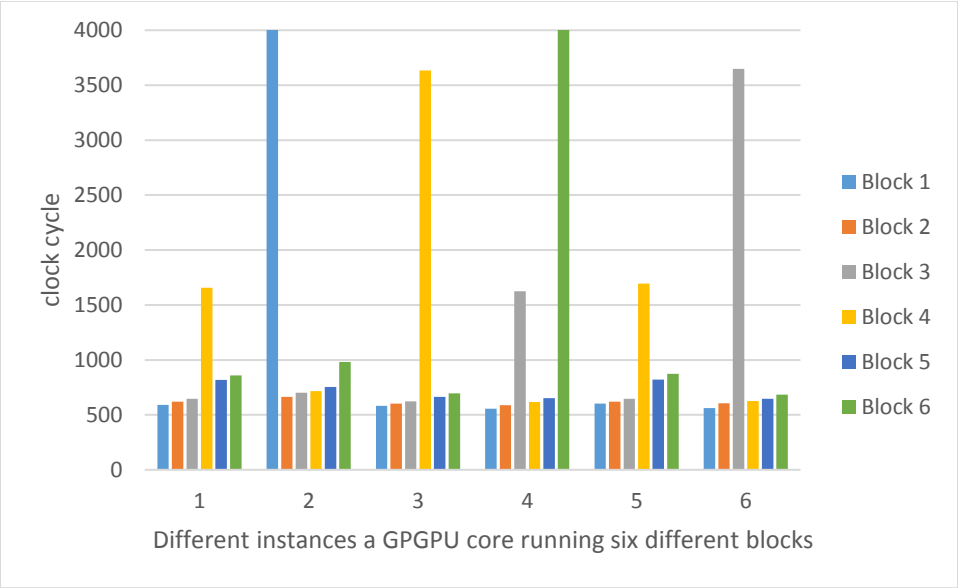

Figure 2: Execution time of different thread blocks running on GPU core

By analyzing execution time of each thread block running on every GPGPU core we can determine the ones running slower which may be offloaded to the PIM cores. Offloading these blocks will not only speedup the performance but also ensure more efficient resource utilization across both GPU and PIM cores.

3. Background

3.1 General Purpose Graphics Processing Unit (GPGPU)

GPGPU units are use of GPU, which are typically used for computer graphics processing, to perform complex computations traditionally handled by a Central Processing Unit (CPU). Modern graphics architectures have become very flexible to program as well as very powerful with high computation speed, increased precision and rapidly expanding programmability of hardware. These features have made GPU an attractive platform for general purpose computing [4].

3.2 3D-stacked DRAM memory

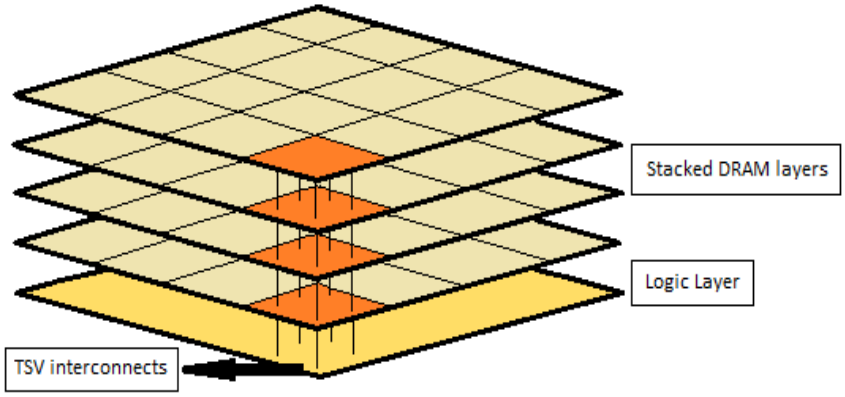

Figure 3: 3D stacked memory concept with logic layer and TSV interconnects

3D stacking brings the primary advantage of increased memory density over conventional 2D designs by stacking multiple DRAM dies atop of each other on a single chip. Stacked memory technology promises memory access with low latency, high bandwidth and lower power consumption [28]. Another advantage with stacking technology is that the base die can be utilized to integrate controllers and high speed signaling circuits hence bringing them much closer to memory [2]. 3D stacking has

been possible in recent years only due to development of *Though Silicon-Via* (TSV). TSV technology facilitates vertical interconnects between the stacked DRAM dies hence providing a shortest possible way to connect two DRAM dies [40, 41]. TSV based vertical interconnects have proven to have very low latency, high bandwidth and energy efficient data transfer between dies in a package.

3.3 Processing in-Memory

In *Processing in-Memory* (PIM) technology, computational/logic units can be placed very close to the memory to achieve faster memory access. Integrating processing logic with stacked DRAM memory can help utilize the high internal memory bandwidth. PIM has been possible only due to inventions in 3D stacking technology [3] as discussed before. Stacking technology has enabled to incorporate memory and logic very close to each other, hence providing dense and fast memory interactions. In our work, we consider *Hybrid Memory Cube* (HMC) [42] as our primary reference platform for implementation and integration of PIM with GPGPU.

3.4 Hybrid Memory Cube

Hybrid Memory Cube (HMC) is a concrete example of current advances in die stacking technologies. HMC is a closed bank memory architecture with four or eight DRAM dies and one logic die (at the base) stacked together using TSV technology. This design improves bandwidth, latency and energy characteristics — without changing the high-volume DRAM design currently used in various systems. HMC storage paradigm is developed by a consortium of memory industry manufacturers and consumers. The *Hybrid Memory Cube Consortium* (HMCC) is backed by several major

technology companies including Samsung, Micron Technology, Open-Silicon, ARM, Altera, and Xilinx.

The logic layer in HMC contains several memory controllers which communicate with the memory storage elements in the DRAM dies using TSV interconnects. Within HMC, memory is organized into vaults. Each vault is functionally and operationally independent. Each vault has a memory controller (called a vault controller) present in the logic base that manages all memory reference operations within that vault. Each vault controller can determine its own timing requirements. Refresh operations are controlled by the vault controller, eliminating this function from the host memory controller. A vault can be thought of roughly equivalent to a traditional DDRx channel since it contains a controller and several independent banks of memory that all share a bi-directional data bus.

Capacity is a clear benefit of HMC architecture. Currently, 4 DRAM die stacks have been demonstrated by Micron and future plans to stack 8 dies has been mentioned. Furthermore, multiple HMC cubes can be chained together to further increase the capacity. However, this will be limited by latency and loading considerations [9].

4 Modelling PIM enabled GPGPU

4.1 GPGPU Simulator

In this chapter we discuss, in detail, about the GPGPU simulator system, memory hierarchy, and its interactions with the GPGPU cores.

4.1.1 Background on GPGPU-sim

The general purpose computations on GPU is modeled using a widely used GPGPU-sim 3.x [19], a cycle level GPU performance simulator to model GPU computing. GPGPU-sim 3.2.0 is the latest version of GPGPU-sim. This simulator models GPU microarchitectures similar to those in NVIDIA GeForce 8x, 9x and Fermi series. The intension of GPGPU-sim is to provide a substrate for architecture research rather than to implement an exact model of any particular commercial GPU, hence making it central to this research. The GPU modeled by GPGPU-sim is composed of Single Instruction Multiple Thread (SIMT) cores connected via an on-chip connection network to memory partition units that interface to graphics GDDR DRAM. We simulate all the benchmarks with default GPGPU-sim's configuration (for steaming multiprocessor, shader processors, and caches) that closely models a NVIDIA GTX 480 chipset.

4.1.2 GPGPU-sim memory organization

The following figure 4 gives a top level view of the organization used by GPGPU-sim. The simulator uses a GDDR DRAM. Each DRAM is interfaced with one memory

partition unit. The memory system is modelled by a set of memory partitions which effectively function as memory controllers to each DRAM die connected to it.

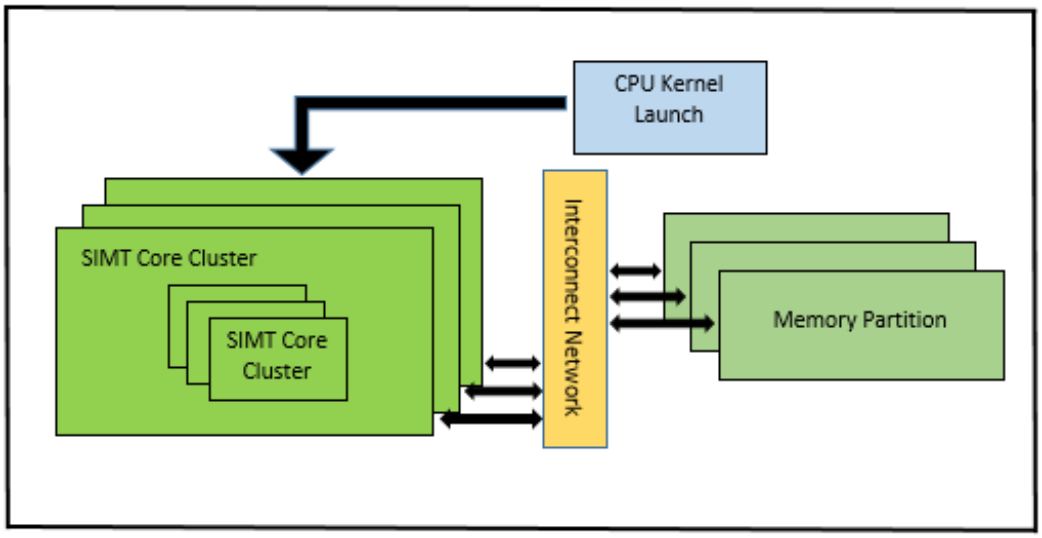

Figure 4: Overall GPU architecture modeled by GPGPU-sim [19, 20]

In GPGPU-sim, the memory partition unit is responsible for atomic operation execution, address decoding and it also contains L2 cache. In addition to these three sub-component units, memory partition also contains various FIFO (First-In-First-Out) queues which facilitate the flow of memory requests and responses between these sub units.

The DRAM latency queue is a fixed latency queue that models the minimum latency difference between a L2 cache access and DRAM access. This component of the memory partition unit is used to implement the TSV latency in HMC. DRAM scheduling is implemented using two different page models: a FIFO queue scheduler and a FR-FCFS (First-Ready First-Come First-serve) scheduler.

The FIFO scheduler services requests in the order they are received. However, this tends to cause a large number of pre-charges and activates and hence may result in a

poor performance especially for applications that generate large amount of memory traffic relative to the amount of computation they perform.

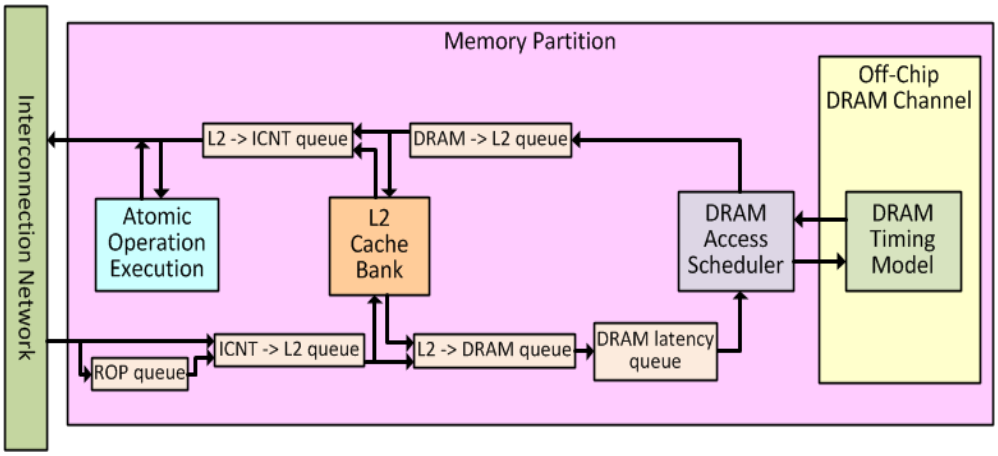

Figure 5: GPGPU-Sim memory partition component [19, 20]

The First-Ready First-Come-First-Served (FR-FCFS) scheduler gives higher priority to requests that will access a currently open row in any of the DRAM banks. The scheduler will schedule all requests in the queue to open rows first. If no such request exists it will open a new row for the oldest request.

4.2 Stacked Memory Organization

We implement a 3D stacked memory which utilizes direct stacking of DRAM dies on a processing chip. We closely follow the stacked memory architecture provided by the HMC consortium in our work. A concept of HMC memory organization is shown in the figure 6. The HMC memory is organized into vaults where each vault is functionally and operationally independent from the other. This allows parallel access similar to DIMM structure for DRAM. Each vault consists of its own memory controller which is known as vault controller in HMC's vernacular.

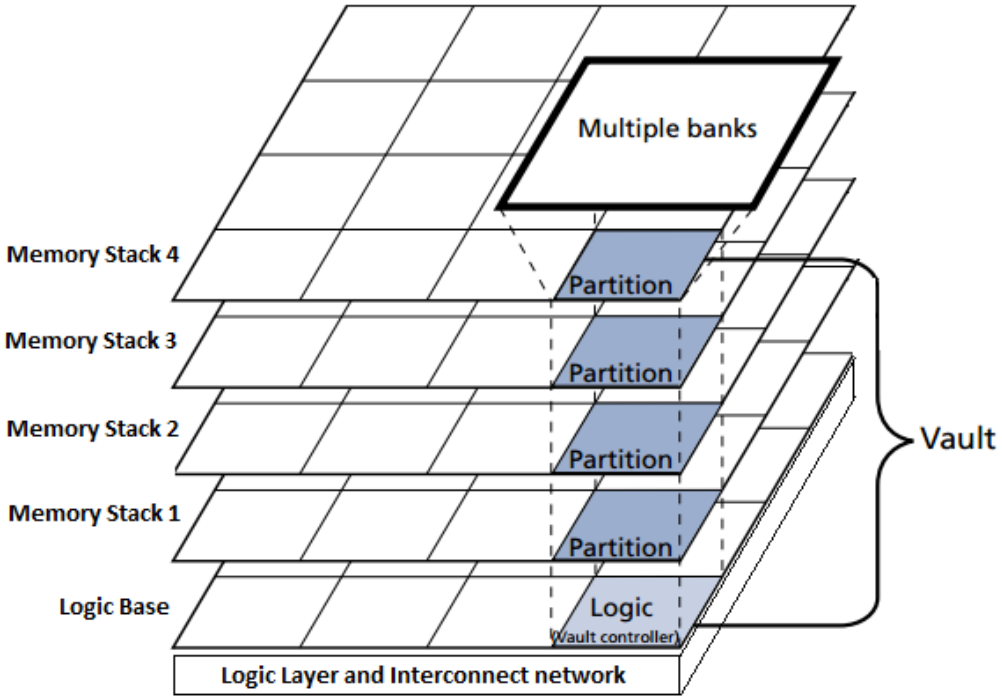

Figure 6: HMC memory and in-memory processing logic organization [5]

The vault controller is capable of handling refresh operations and manage all memory operations with its vault. High speed serial links are used for communication between the HMC and the host GPU. The packets coming from the host via links to the HMC for read/write requests are routed to their respective vault controllers using the interconnect network. The write acknowledgements and read data follow the same path back to the GPU.

The DRAM dies in the stacked memory organization are connected by a dense interconnect of Through-Silicon Vias (TSVs). These are metal connections that extend vertically through the entire stack. These interconnect path between the stacks are very short with lower capacitance than long PCB trace buses. Hence the data can be sent at a very high data rate on TSVs without having expensive and power hungry I/O divers.

Furthermore, to increase parallelism in HMC stacked architecture, the dies are segmented into vertical vaults. These vaults contain several partitions and each of these partitions contain several banks similar to a DRAM organization.

According to HMC specifications [5], a single HMC can provide up to 320 GB/s of external memory bandwidth using eight high speed off-chip serial links (SerDes links). On the other hand, HMC is also capable of providing an aggregate internal bandwidth of 640 GB/s per cube, which consists of 16 vaults per cube. Each vault in the HMC is analogous to a DRAM partition unit.

Table 1: HMC configuration

Configuration	
Number of links per package	2, 4 (SerDes Links)
Link lane speeds (Gb/s)	12.5, 15, 25, 28, 30
Memory density	2GB, 4GB (under development)
Number of vaults	16, 32
Memory banks	2GB: 128 banks 4GB: 256 banks
Maximum aggregate link bandwidth	480 GB/s
Maximum DRAM data bandwidth	320 GB/s
Maximum vault data bandwidth	10GB/s

4.2.1 Integrate stacked memory with GPGPU

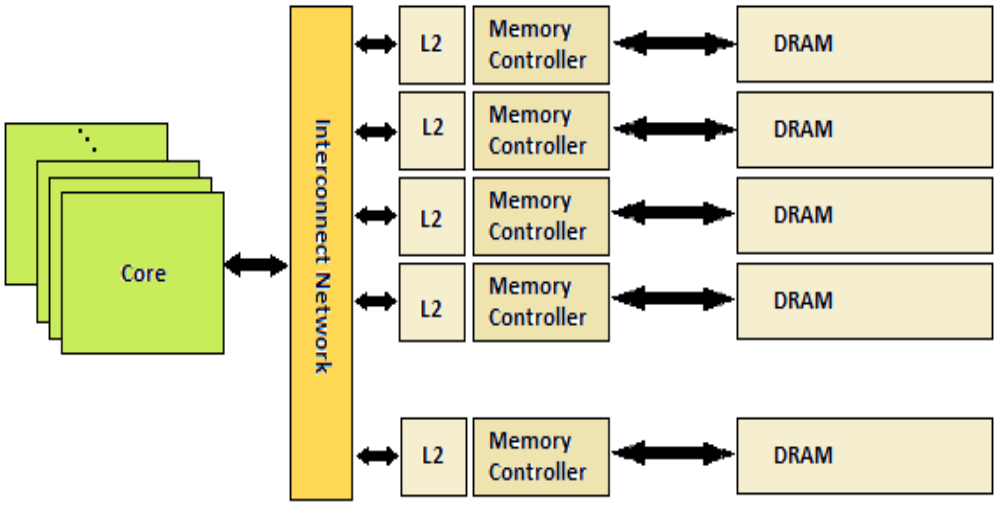

Figure 7: GPGPU Sim memory architecture overview [19]

We integrate stacked memory with GPGPU by replacing GDDR memory used by GPGPU with HMC's memory architecture. Figure 7, shows the overview of memory architecture used in current GPGPU.

In GPGPU-sim memory organization each memory controller is connected to a single DRAM die. This connection can be thought of as organized vertically to implement one vault in HMC. Each of these vaults are functionally and operationally independent and each vault controller determines its own timing requirements. Refresh operations are controlled by the vault controller i.e. memory controller in GPGPU-sim.

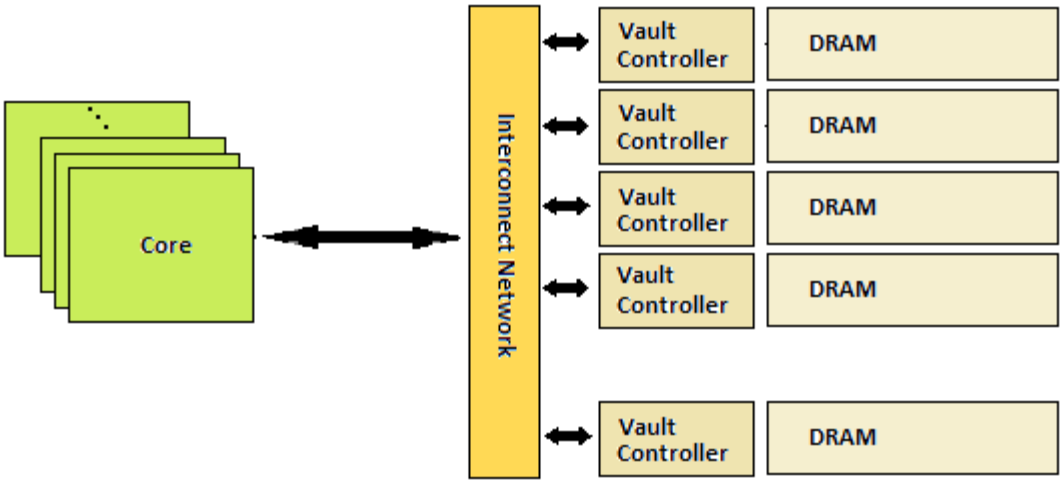

Figure 8: HMC memory architecture implemented in GPGPU-sim

Vault controller buffers are implemented by FIFO queues available in the memory partition unit. The vault controller also has the ability to schedule references within a queue completely out-of-order rather than by the order of their arrival.

The access between the DRAM dies is facilitated by the TSV interconnects. Latency of this access is taken as 1.5 nsec or 1 cycle for a 667 MHz bus frequency [16]. This latency is considering the worst case situation, when the data is to be accessed from the vault controller to the topmost DRAM layer in a stack of 8 DRAM dies. The access latency caused by the TSVs is however very small as compared to the latency of 100 cycles to access data from off-chip main memory. In GPGPU-sim, DRAM latency queues is used to implement this access latency overhead of TSV (requests in the DRAM latency queue wait for a fixed number of SIMT core cycles before they are passed to the DRAM channel). The high bandwidth data transmission provided by the TSV channels is implemented in GPGPU-sim by changing the DRAM operating

frequency in GTX 480 configuration. Figure 8 illustrates the conceptual view of the stacked memory architecture implemented on GPGPU-Sim.

The HMC architecture contains quite a large design space to be explored in order to optimize the performance. Within the DRAM memory stack we deal with specific design factors to expose a proper level of memory parallelism (implemented by vaults in HMC) to effectively utilize the available TSV and off-chip link bandwidth. In the DRAM stack we have 2 fixed resource configuration (number of DRAM dies stacked and number of banks on each die): 128 and 256 bank configuration. Both of these configurations consist of 16 vaults and four or eight DRAM dies. TSV bandwidths available are 20, 40, 80, 160 and 320 GB/s [9]. The maximum bandwidth exploitable by the core is 640 GB/s [21].

4.2.2 Off-chip links

Communication between the stacked memory and the GPGPU is provided by the off chip links. HMC architecture provides different choices of off-chip link bandwidth available from the stack memory to the host GPGPU. These links consist of multiple serial lanes with full duplex operation used for communication between the GPGPU processor cores and the stacked memory. The links transmit commands and data, in both direction, enclosed in packets called “FLITs” which consist of a fixed number of bits (according to the HMC-2.0 specifications, a flit size is 128 bits). Raw link bandwidth configurations available are 80, 160, 240, and 320 GB /s [9].

4.3 Processing In-Memory

Processor architecture for host GPGPU and in-memory cores in our system organization are in-order processing units which use Single Instruction, Multiple Thread (SIMT) model. In GPGPU-sim terminology, each processing unit is known as a shader core which is similar in scope with *streaming multiprocessor* (SM) in NVIDIA terminology [19]. We believe choosing SM as in-memory processor has several benefits, firstly, by using existing GPU design we ease the task of redeveloping dedicated processing units for PIM and promote reusability of off-the-shelf technology. Secondly, programmability of existing SMs in GPUs will provide a broad range of applications to completely utilize PIM [1]. Most importantly, using SM will provide a uniform processing architecture [1].

The shader cores used in both GPGPU and PIM has a SIMD width of 8 and uses a 24-stage, in-order pipeline without forwarding [19]. The pipeline consists of six logical pipeline stages which are fetch, decode, execute, memory1, memory2, and writeback. Post-dominator re-convergence mechanism is used to handle branch divergence in threads. Figure 9 and 10 shows a detailed implementation of a single shader core in GPGPU [19] and the in-memory core that we used for PIM. The cores in PIM implement the same thread scheduler and SIMD pipeline, however, the PIM cores do not have access to L1 cache units, L2 cache and shared memory. This is done to analyze PIM cores strictly with stacked memory and without the benefits from caching.

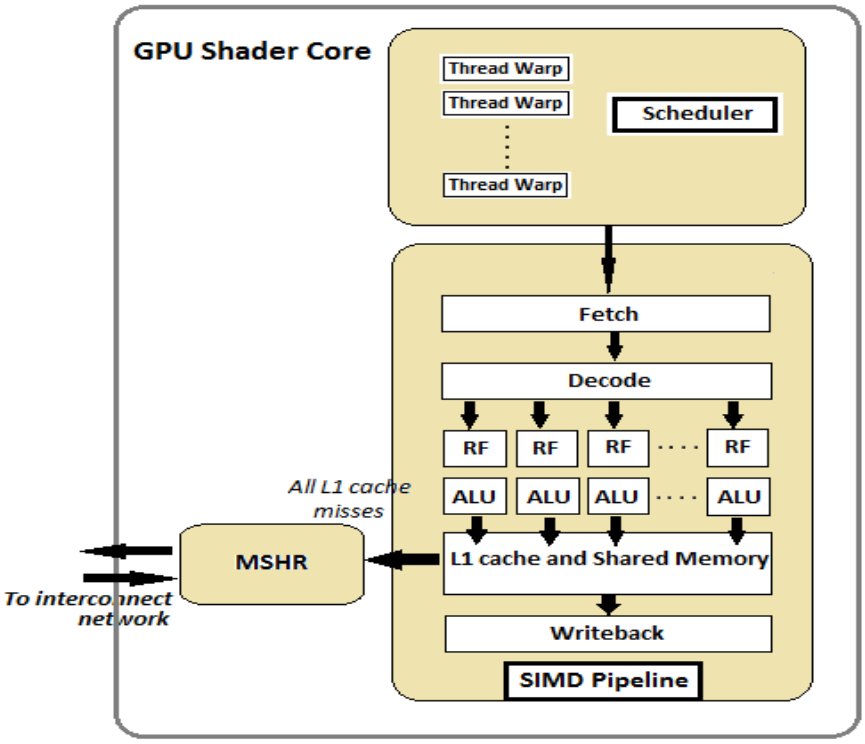

Figure 9: Detailed GPU shader core

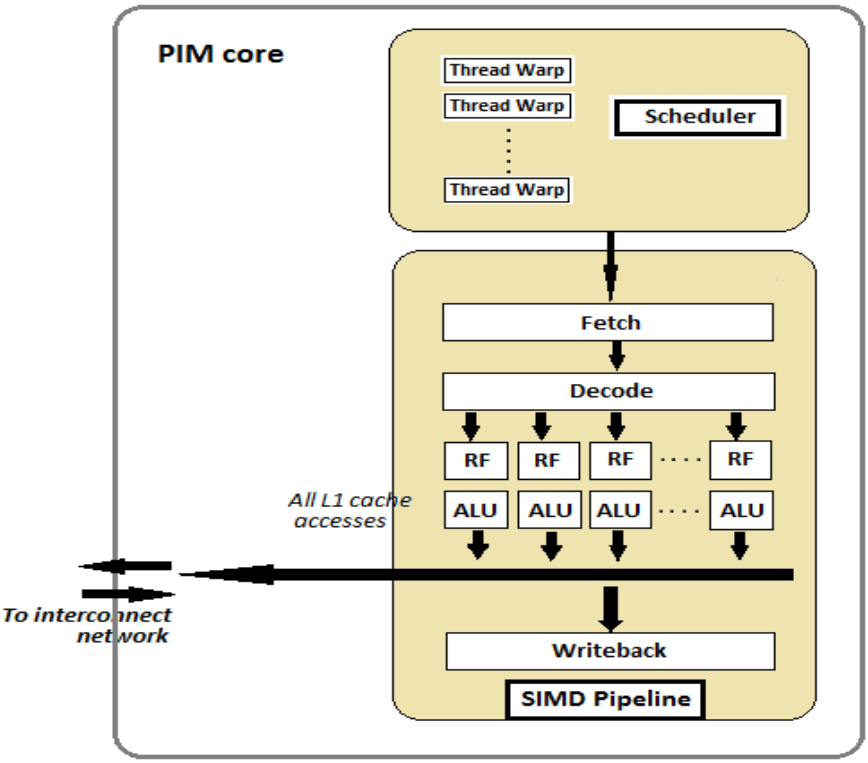

Figure 10: Detailed PIM logic core

Furthermore, having L1 cache with PIM cores would complicate management of cache coherency across GPGPU and PIM caches [6]. Deploying traditional cache coherency protocols on our system with so many core would potentially require additional states and would also consume significant part of off-chip link bandwidth.

5. Block Offloading Mechanism

In this chapter we describe our new runtime mechanism that records and analyzes average execution time of all the thread blocks running on each GPGPU cores. It then dynamically determines possible thread block candidates for offload and then decides at runtime whether the selected block candidate should really be offloaded to PIM cores.

5.1 Identifying block offload candidate

The main objective when identifying candidates for offload is to find which thread blocks need higher memory bandwidth and offload these blocks to improve performance and resource utilization in both GPGPU and PIM. Primarily, a block of threads could be considered for offload to PIM if its execution is taking much longer time to finish as compared to other thread blocks running on the same core. This comparison is made between the blocks which were initialized/launched at the same time and on the same GPGPU core. Number of times the other thread blocks have completed execution further strengthen the identification of slower thread blocks which can be offloaded.

Increased execution time of a thread block could be due to multiple factors such as high thread divergence in the block or some of the threads in the block are waiting at a barrier or high cache miss rate causing frequent accesses to the main memory. The third case is the one we are most interested in. L1 data cache miss rate and *miss status holding register* (MSHR) are the main indicator of this case. So, if the execution time of a block candidate on a GPGPU core is found to be higher than other blocks (running on the

same core) and L1 cache miss rate of that core is higher than a threshold miss rate value, then this block becomes a possible candidate for offload to the PIM cores. Chapter 5.4.1 further explains the implementation in detail and describes the steps to identify a slower block and determine whether that block is actually offloaded depending on network traffic, availability of PIM cores to service the offloaded block and number of active warps in the block.

5.2 Block offload limitations

Few limitations to offload candidate blocks are –

1. If the candidate block has divergent threads, then they must first converge before being offloaded. Trying to offload a divergent thread block adds large amount of complexity for managing SIMT stacks which control the divergence and re-convergence information GPGPU system.
2. If any warp in the candidate block is waiting at a barrier, then all threads/warps must reach the barrier before the block can be offloaded.
3. Any warp in the block must not already be issued before offload. If any warp is issued, then its SIMT stack information may change or the threads in the warp may diverge. In GPGPU-sim, once a warp is issued, it is functionally simulated by cuda-sim simulator in the same cycle, hence the execution (issue stage in GPGPU sim) might cause divergence and will certainly change the SIMT stack for that warp.
4. No warp in the offload candidate block must have a pending store acknowledgement. Since the store acknowledge comes back at the same core that requested it, offloading the block before it receives the acknowledgement would mean

that now the block is present elsewhere in the system. Without receiving the store acknowledgement, the threads cannot finish their execution resulting in a deadlock situation.

5. There should not be any shared memory access within the candidate block. Since PIM does not implement a shared memory, the threads in that case will have to make access to the shared memory present on the host GPGPU system through the off-chip links.

6. The candidate blocks must have at least two warps active at the time the block is offloaded. This is because if a thread block has, for example, only one active warp, then there is a possibility that the warp might be near completion. In that case, offloading the block would simply add extra cycles to its execution time without any significant performance gains.

5.3 Block offload aggressiveness

Block offload aggressiveness is a very important factor to ensure proper resource utilization across the system. Offloading blocks very aggressively can lead to couple of issues in the system. Firstly, offloading very frequently can increase traffic on the off-chips links and the interconnect network significantly, making them a new bottleneck. Second, offloading blocks to PIM when they are already fully occupied by other blocks will make the offload block candidate wait for the PIM resources to become available. This is because each SM has a limit on concurrent blocks running on it. This limitation is imposed by the GPGPU hardware. On the other side, if the offload aggressiveness is too low then most of the PIM cores will be left underutilized.

Hence offload aggressiveness need to be tuned to a certain value to get optimum performance.

5.4 Implementation Details

In this chapter, we describe the implementation of block offloading system. Firstly, we introduce all the components introduced into the system and then we provide a detailed description of block offloading (chapter 5.4.1).

5.4.1 GPGPU pipeline with block offloading

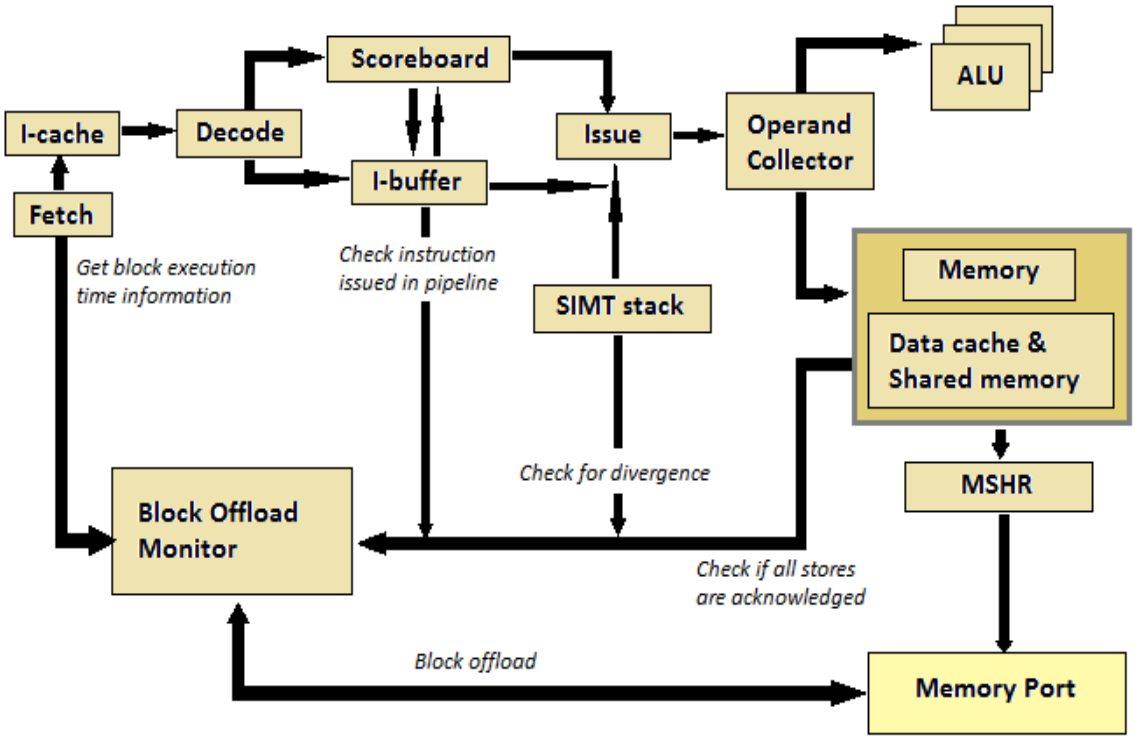

Figure 11: Block diagram of our offloading system and its interaction with GPU pipeline

Figure 11, shows a high level block diagram of how the block offloading system fits into the GPGPU pipeline [19]. To support block offloading we add a new component namely *block offload monitor* (BOM). Block offload monitor maintains and

records the execution/completion time (in clock cycles) of each block running on each of the GPGPU cores. It then decides, at runtime, whether to offload the block to PIM through the memory port.

Block Offload Monitor (BOM)

The Block execution monitor maintains and records average execution/completion time of all the active blocks. It then selects a candidate blocks for offload using this runtime information. Analysis is made based on the following two cases –

(1) If the BOM finds a block running much slower than the others. Details of this analysis is described in the figure 12.

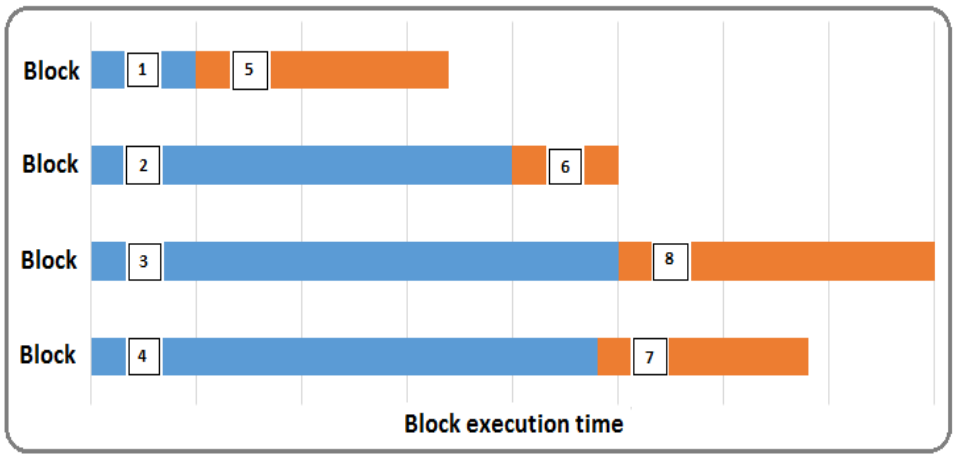

Figure 12: Block execution times for different blocks running on same core

In the above figure, thread blocks 1 to 8 are available for execution on GPGPU cores. Assume that blocks 1, 2, 3 and 4 are launched first and blocks 5, 6, 7, and 8 are waiting to be scheduled. Among the four blocks, block #2, #3 and #4 are running slow and take significantly more number of clock cycles to complete execution than block #1. To identify this, the BOM needs to wait till block #1 i.e. the fastest block to finish

execution. Once block #1 is completed, the BOM will have a frame of reference against which it will check to identify whether other active blocks are running slow or not.

Once block #1 finishes execution, the BOM checks for multiple number of factors to determine if the other active blocks will take longer time to execute. These checks are important because there could be a situation, see figure 13, where block #2 might finish execution right after block #1. In that case, block #2 should not be selected for offload. Hence the final decision to offload a block is made by the BOM after performing checks for such situations.

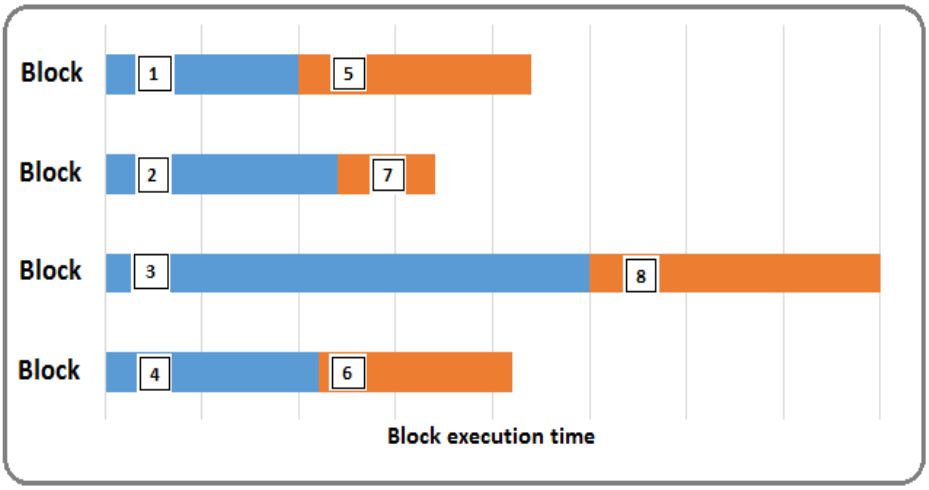

Figure 13: Block execution monitor senario-2

(2) This case can occur when all the blocks on a core are running slow. Here the average execution time and number of active warps in the block helps the BOM to select candidate blocks for offload to the PIM. If the execution time of a block executing in a GPGPU core is higher than the average execution time of all the previously executed blocks on the same hardware GPGPU core, then that thread block becomes a candidate for offload to the PIM cores. However, the selected thread block is actually offloaded

only after making few more runtime condition checks. These checks are made to ensure that the offloading the selected thread block will give performance speedup. Firstly, the number of active warps in the thread block must be greater than a statically determined threshold which is two active warps in a thread block.

It then checks the L1 data cache miss rate against a dynamically determined threshold miss rate value. The L1 data cache miss rate of the GPGPU core from which the block is being offloaded must be greater than the threshold value. Dynamic threshold for L1 cache miss rate is determined by calculating the average miss rate of all the GPU cores at that instance. After checking for L1 cache miss rate, next the candidate block is checked if it has any pending store acknowledgements, if any warps in block candidate are divergent, and none of the warps in the candidate block must be issued or waiting at a barrier at the time of offload (chapter 5.2).

After all the runtime checks pass, the candidate block is offloaded only if there are hardware resources available at the PIM. If the block is not offloaded, then the block execution resumes on the same GPGPU core. If the block is offloaded, the BOM sends all the register values associated with each thread in the block, SIMT stack of each warp which holds the active threads and the program counters. In our work we assume latency of offloading a block is 3 core clock cycles. This assumption is conservative as compared to the instruction offload latency which involves creating and offloading new child kernels [6, 37]. The penalty of block offloading is however much smaller than that of hundreds of clock cycles for each main memory access. After offloading, the GPGPU cores can continue execution of other thread blocks waiting in the scheduler.

5.5 Simulation setup and evaluation

5.5.1 Methodology

We model our system organization by modified GPGPU-Sim 3.2.0 [19]. We model and integrate stacked memory and PIM logic layer with GPGPU-Sim. Off chip links that provide connection between the GPU and stacked memory is modelled as high speed unidirectional SerDes links [41] using *Intersim* [43], a modified version of *Booksim* [44]. Booksim is a cycle accurate interconnection network simulator used for on-chip interconnect simulation. However, unlike booksim which supports only a single interconnection network, intersim can simulate two unidirectional interconnection network: one for traffic from GPU to memory (uplink) and one network for traffic from memory to GPU (downlink).

In our PIM system organization, we assume inter stack access to allow the PIM core of one vault to access memory location present in other vault. This is facilitated by the interconnect network present in between the stacked memory and the logic layer, and should not incur any performance loss due to high internal memory bandwidth provided by the HMC architecture [17, 41]. We assume the internal memory bandwidth provided by the memory stacks is 2x of that available to the links between the GPGPU and the memory stacks. This assumption is in accordance with the HMC consortium and reputed prior works in this domain [1, 6, 21, 38]. We also evaluate the system with same internal and external memory bandwidth so as to see that the PIM is not just advantageous because it can exploit more bandwidth than the GPGPU (chapter 5.5.5).

Table 2 provides the configurational details of our simulated system. To make a fair comparison between the baseline GPGPU and PIM enabled GPGPU (GPGPU+PIM) system, we take the same number of processing cores across both the systems. We consider NVIDIA's GTX 480, a state of the art GPU architecture, as the baseline GPU in our work.

Table 2: Configuration setup

Processing units	
Core number	15 cores for GPU 8 cores for PIM
Core clusters	1 core per cluster
Core configuration	1400 MHz core clock frequency, 48 warps/SM, 32 threads/warp, 32768 registers, 8 CTAs/SM, 48 KB Shared memory (GPU cores only)
Private L1 cache per core (GPU only)	32KB, 4-way, write through
Shared L2 cache (GPU only)	1MB, 16-way, write through, 700 MHz
Scheduler	Greedy Then Oldest (GTO) dual warp scheduler, Round Robin (RR) CTA scheduler
Links and interconnect	
GPU to memory (off chip links)	80 GB/s per link, 320 GB/s total (4 links)
Interconnect	1.25 GHz clock frequency
Memory stack	
PIM to memory (internal memory bandwidth)	10 GB/s per vault, 160 GB/s per stack, 640 GB/s total for 4 stacks.
Configuration	4 memory stacks, 16 vaults/stack, 16 banks/vault, 64 TSVs/vault, 1.25 Gb/s TSV signaling rate.
DRAM timing	GDDR5 timing from Hynix H5GQ1H24AFR
DRAM scheduling policy	FIFO

Table 3 shows the benchmark applications we use to evaluate our system. This are memory intensive workloads from Rodinia 3.0 [18], and GPGPU-sim [19]. All the

applications are run till completion and performance metrics, provided by GPGPU-Sim at the end of execution, are recorded for each application.

Table 3: Applications used

Application Name	Domain
Breadth-First Search (BFS)	Graph Algorithm
LIBOR Monte Carlo (LIB)	Numerical Algorithm
3D Laplace Solver (LPS)	Finance
MUMmerGPU (MUM)	Graph Algorithm
N-Queens Solver (NQU)	Back tracing
Ray Tracing (RAY)	Graphics Rendering
StoreGPU (STO)	Hashing
Braided B+tree	Tree Algorithm
Back Propagation (backprop)	Pattern Recognition
Gaussian	Scientific

Power model methodology

To evaluate energy consumption of PIM enabled GPGPU system, we use GPUWattch [22]. GPUWattch is an energy model based on McPAT which is integrated into GPGPUSim – 3.2.0. We model power consumption of the processing cores, off-chip links and the DRAM stacked memory.

For the power consumed by the off-chip serdes links, we assume the sum of transmit and receive energy per bit when real data is being transmitted ($E_{bit,real|pkt}$) to be 2.0 pJ/bit. Energy consumed by the links when no data is being transmitted, i.e. when the link is idle ($E_{bit,idle|pkt}$) is assumed to be 1.5 pJ/bit [24]. This is because high-speed signaling requires sending/receiving idle packets over the channels even when there is no data to communicate.

The power consumed by the links can be generalized as -

$$Network\ energy = E_{bit,real|pkt} * D_{real} + E_{bit,idle|pkt} * D_{idle}, \quad (1)$$

where, D_{real} (D_{idle}) is number of packets in flits transferred through the link when data (no data) is transferred. Using this equation, we calculate the power consumed (worst case) by four SerDes links to be 5.04W.

Now, to calculate the power to access DRAM arrays for both read and write accesses, we use the following equation,

$$Power\ to\ access\ DRAM\ arrays\ (read\ \&\ write) = E_{bit,\ DRAM\ layer} * (burst\ length * bus\_width) * (\#reads + \#writes), \quad (2)$$

here, energy per bit for DRAM layers ($E_{bit,\ DRAM\ layer}$) = 3.7 pJ/bit, burst length = 8, bus_width = 16 bytes for HMC, 4 bytes DDR3

Energy per bit for logic layer ($E_{bit,\ logic\ layer}$) = 6.78 pJ/bit.

Hence, total energy consumed by the HMC logic layer and the DRAM layers will be 10.48 pJ/bit [17].

5.5.2 Performance results

In this chapter, we evaluate the performance improvements obtained from the PIM enabled GPGPU and the PIM enabled GPGPU with block offloading monitor against the baseline GPGPU by analyzing the effects across 10 benchmarks (table 3). All the results have been normalized to the baseline GPGPU architecture. Figure 14 shows the performance results for each benchmark: 1) baseline GPGPU, 2) PIM enabled GPGPU

(GPGPU + PIM), 3) GPGPU and PIM with block offloading using BOM. Number of processing cores i.e. SMs, are kept same across all the schemes and number of memory partition units (memory channels) in the baseline GPU is also kept same as the number of vaults in the stacked memory. This is done to ensure fair comparison by maintaining equal number of resources.

With PIM enabled GPGPU (GPGPU + PIM), we observe a performance improvement of 36% on an average. When block offloading using BOM is enabled with all the restrictions imposed (chapter 5.2), speed up in performance is 30% on average. Here we can see that PIM enabled GPGPU clearly outperforms the baseline architecture. Performance with block offloading using BOM suffers 6% as compared to GPGPU + PIM without BOM scheme. This is because of the restriction placed in BOM i.e. blocks with divergent threads cannot be offloaded to PIM. Due to this restriction, the PIM cores are not always completely occupied with thread blocks. Whereas in GPGPU + PIM without BOM, blocks are directly issued to PIM cores without any restrictions (described in chapter 5.2) which follows an ideal situation. This ensure complete utilization of PIM cores. However, without BOM, non-memory intensive blocks are also offloaded to PIM cores. Therefore, even if the PIM cores are completely utilized in GPGPU + PIM without BOM, its performance is not much greater than GPGPU + PIM with block offloading using BOM.

Block offloading system is very efficient in recognizing and speeding up applications with high global memory accesses, this can be further observed from the performance results. Maximum performance improvements are seen with memory intensive applications: 2.1x improvement for BFS, 75% improvement for MUM and

30% speed up with LIB and b+tree. This is mainly due to heavy global memory traffic generated by these applications that can be effectively satisfied by the internal memory bandwidth from stacked memory. Furthermore, high L1 cache miss rate with these applications makes the process of identifying candidate blocks for offload much more efficient.

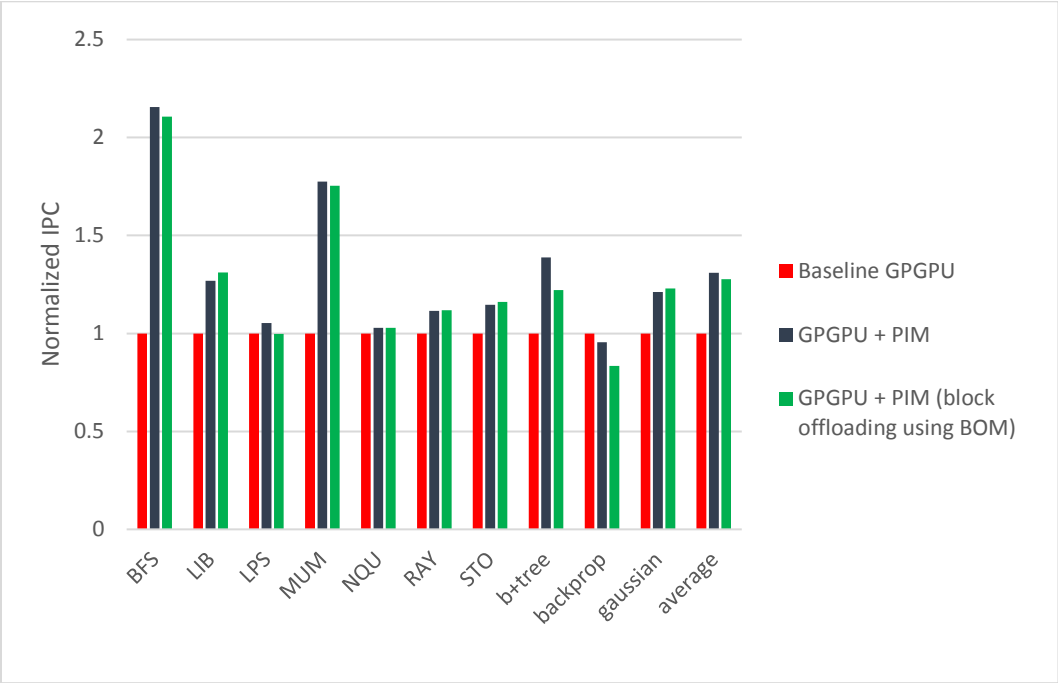

Figure 14: Performance speed up comparisons

In applications LPS and STO, performance improvements are 4% and 16% respectively compared to the baseline GPGPU. This is due to the fact that these applications have been optimized to use shared memory resources [19], hence they do not benefit much from block offloading system. However, LPS and STO still have a fair number of coalesced global memory accesses which accounts for the performance improvements. Performance of NQU is same as the baseline architecture, this because of very high divergence caused by a single thread which performs most of the computations. The only application that suffers from our system is backprop (*back*

propagation). This is mainly because backprop works on an unstructured grid [18], hence it benefits from the locality provided by caches which are not available on the PIM cores in our system.

5.5.3 GPU resource utilization

Figure 15 shows the average block execution times of all the thread blocks running on the host GPGPU cores (for clarity we only show graphs for three benchmarks). The graphs on the left side, in the figure, show block execution times on applications running on the baseline GPU architecture. The ones on the right side, in the figure, represent the same for applications running on our system with block offloading.

Firstly, as mentioned in chapter 2 as one of the motivations for block offloading, our system organization helps to provide a more uniform GPU resource utilization. The average block execution time on the GPU cores are now in much narrower range. This is mainly due to offloading bottleneck blocks to PIM which helps in freeing up resources on the GPU in a more uniform and predictable manner. Secondly, we want to highlight that the average block execution time also has effectively reduced for each benchmark shown in the figure. This is in direct correlation with increased performance with block offloading.

5.5.4 Stalls and bottleneck analysis

In this chapter we observe the effect of block offloading technique on the number of stalls experienced by warps in pipeline. These stalls are caused due to warps waiting for data from the global memory. From figure 16 we can see that the number of stalls

in all applications except STO, b+tree and backprop, have decreased significantly (up to 50% of the stalls in baseline). This is because memory intensive blocks in these applications are effectively identified and offloaded which reduces the number of stalls experienced.

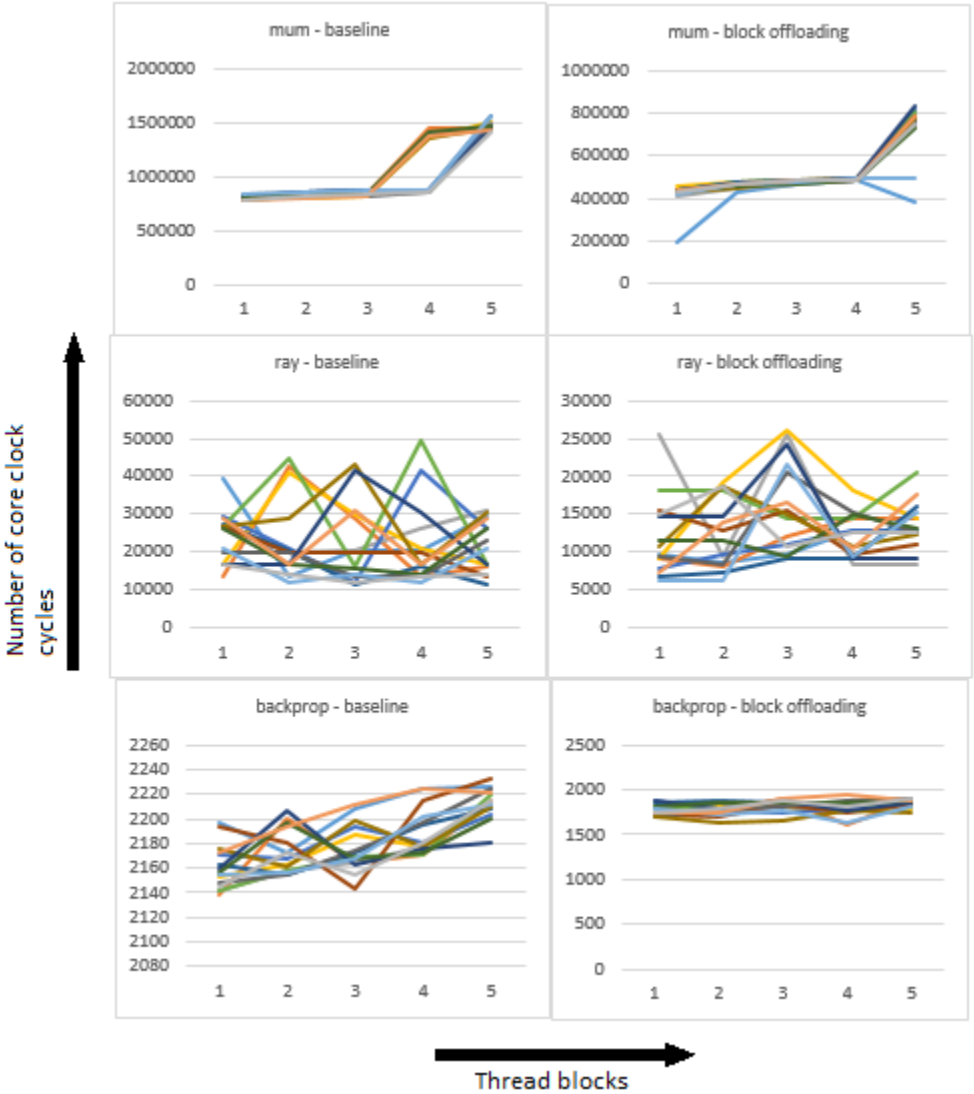

Figure 15: Average block execution time for all the GPU cores with and without block offloading. The y-axis represents the number of clock cycles

In case of STO and backprop, memory accesses in these applications are highly coalesced, hence stalls are introduced due to unavailability of cache resources to offloaded blocks on PIM cores. For b+tree, further evaluations show that the interconnect network connecting to the DRAM becomes a bottleneck adding the stalls experienced. This can be reduced by controlling the offload aggressiveness. However, this becomes a trade-off situation as decreasing the offload aggressiveness will reduce stalls by interconnect network bottlenecks but will also reduce performance of the application.

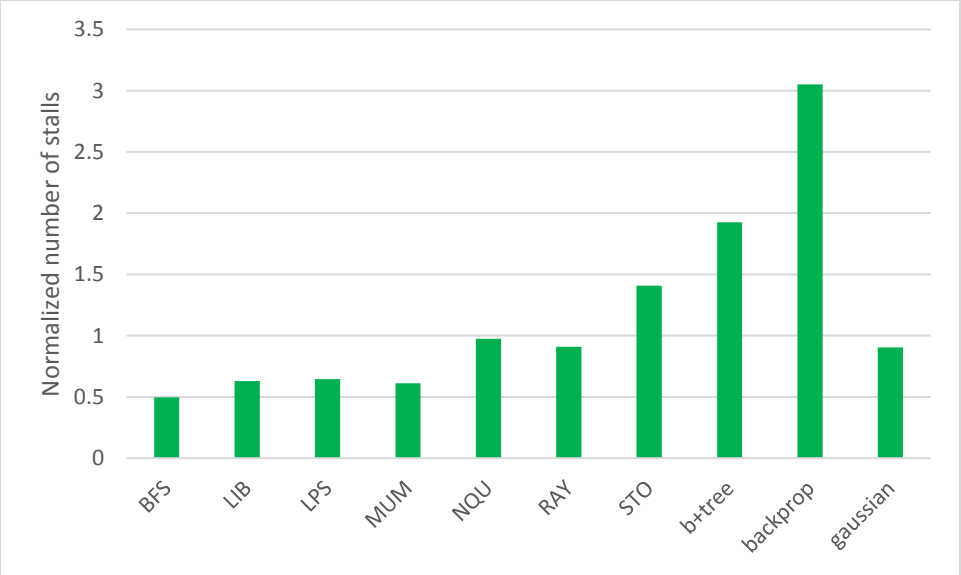

Figure 16: Stalls due to warps waiting for data from memory in block offloading to PIM

5.5.5 Performance dependence on internal/external bandwidth

Figure 17 shows the dependence of performance on the internal and external memory bandwidth. Here we evaluate block offloading with two cases:

i. Internal memory bandwidth is twice as much as external

ii. Internal and external memory bandwidths are equal to each other

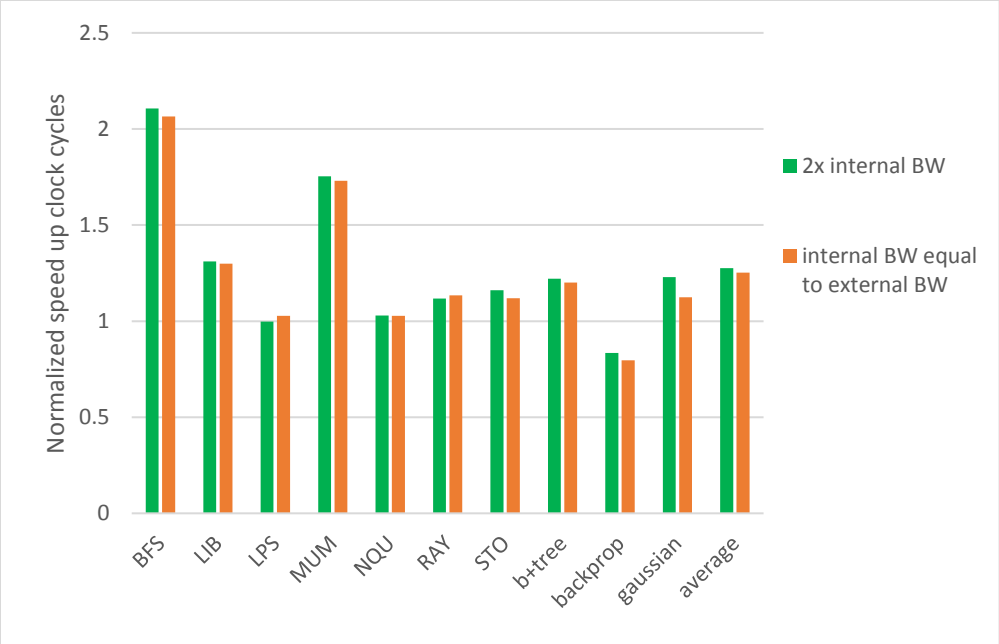

Figure 17: Comparison in performance of block offloading mechanism with different combination of internal and external memory bandwidths

Internal memory bandwidth refers to the total bandwidth provided by the TSVs whereas external memory bandwidth is provided by the off-chip serial links. PIM is often deemed advantageous because the in-memory processing cores can harness this high internal memory bandwidth optimally.

However, from figure 17, we can observe that the performance is not completely dependent on high internal memory bandwidth. In fact we get only around 2.5% performance improvement with internal memory bandwidth twice of that external. This is the case because, GPUs are often bottlenecked by the off-chip links connect GPU to the main memory. Ones the blocks are offloaded to PIM cores, memory accesses from PIM do not have to be constrained by this bottleneck and hence can completely utilize available internal bandwidth form TSVs.

5.5.6 Energy Consumption results

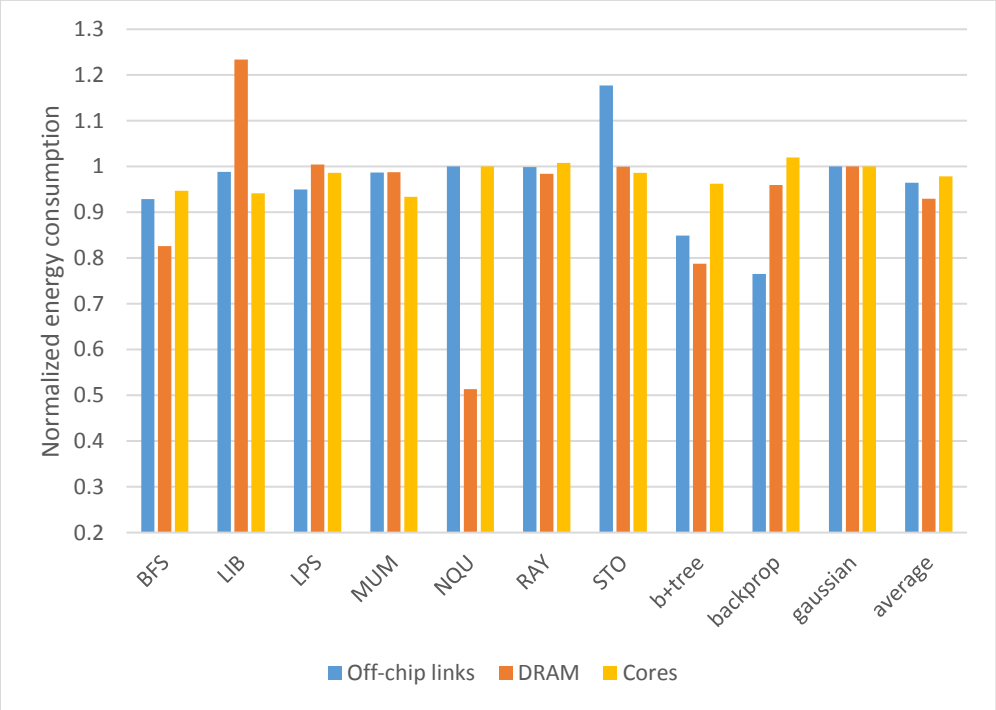

Figure 18: Energy consumption in PIM enabled GPGPU with Block offloading monitor

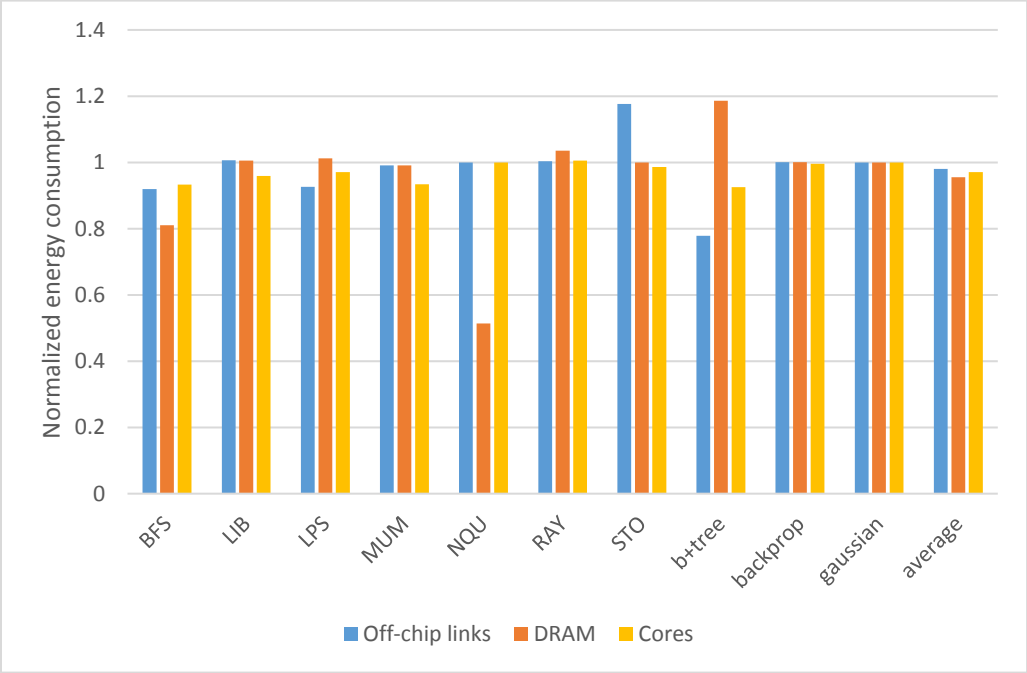

Figure 19: Energy consumption in PIM enabled GPGPU

Figure 18 and 19, shows the energy consumption of the PIM enabled GPGPU as compared to the baseline GPGPU architecture (results are normalized to the baseline GPGPU architecture). We can see that integrating PIM into GPGPU is effective in reducing energy consumption of the system. Total energy consumption is reduced by 5% on an average (maximum of up to 26%). Energy reduction in off-chip links is better in PIM enabled GPGPU with block offload monitor (BOM) because BOM is very effective in identifying memory intensive blocks and offloading them to PIM, hence resulting in lower memory traffic on the off-chip links. Energy reduction in DRAM is 11% on average and maximum of 50%. Reduction in DRAM energy consumption is fairly notable in applications with high number of memory accesses like BFS, b+tree and backprop. This is mainly because of low energy per bit value for DRAM layers in HMC as compared to GDDR used in the baseline GPGPU. Energy consumption in off-chip links is by 5% on average due to reduction in traffic for GPU to main memory accesses. Overall reduction in energy consumption is due to improvements in performance provided by PIM enabled GPGPU. This increased performance reduces the total execution time and hence reducing leakage energy.

6 Related work

6.1 Current DDRx systems

The DDR family of memory is today universally popular with currently DDR3 used in majority of systems boards and DDR4 which is ramping up slowly. DDR4 is expected to replace DDR3 in coming years. DDR4 allows only a single DIMM per memory channel to solve the signal integrity problem been associated with DDR3. By limiting channels to a single DIMM, DDR4 is expected to scale at least to 3.2GT/s (twice the data rate of DDR3 1600) [11]. Although this alleviates the bandwidth problem, it still faces capacity issues. In addition, since a single memory channel requires hundreds of CPU pins, scaling the number of channels to increase channel capacity is not a favorable solution. Therefore, the low availability and high costs of high density DIMMs makes them impractical for big systems. Finally, DDR4 like its predecessors, suffers from the trade-off between bandwidth, capacity and power reductions.

Another likely choice to replace DDR3 and DDR4 is LPDDR4. The LP stands for low power. LPDDR4 is a type of DDR memory that has been optimized for the wireless market. LPDDR's advantages include its widespread adoption and availability and its well defined and stable specifications. The low-power optimization makes it only a little more expensive than DDR, and it still uses the I/O pins that DDR uses. Ease of migration is also a positive factor, because it runs in the same frequency range as DDR. However, the biggest trade-off is its lifetime. Since the wireless market turns over its products approximately every 12 months, LPDDR memories change at a similarly

rapid pace. If a big company sells products for 10–15 years, it is difficult to accommodate a memory device family that changes every 12 months [25].

6.2 2D Processing in-memory

Multiple research teams have built 2D PIM designs and prototypes [26, 27] that confirmed a great potential for speedup in certain applications like media, irregular computations and data intensive applications [28]. Before 2000, there was great interest in processing-in-memory technology or intelligent memory. The designs at that time placed many small general purpose cores n the memory system. These cores were small in order to minimize losses in memory integration, and numerous, to extract high bandwidth [31]. However these works faced a tough road ahead and couldn't make it to commercial markets primarily because the DRAM prices at that time were very low as compared to intelligent memory. However, with the fast growth of chip density PIM is a very promising way to alleviate the memory bottleneck, and possibly the best way to exploit the huge number of transistors available [30].

Recent work in PIM: There are several studies on developing new and more efficient PIM architecture. Pugsley et al. [28] propose a near data computing architecture which focus in-memory processing on MapReduce workloads to utilize parallelism and the largely localized memory access of these applications. SAP HANA in-memory and in-memory database platform [29] is a concrete example that employs a cluster of nodes that deliver an in-memory storage space.

Zhang et al. [1] focus on moving computations closer to the memory to reduce both energy and data movement overheads. However, this approach significantly increases

programmer efforts to identify compute and data intensive nodes to execute it on local host processors or 3D stack logic layer.

Hsiesh et al. [6] proposes work towards a more programmer transparent near data-processing. This approach minimizes programmer efforts and identifies the code blocks for offloading behind the scenes to in-memory processing. Ahn et al. [21] highlights PIM as viable solution to achieve parallel graph processing which is extremely challenging the conventional systems due to severe memory bandwidth limitations. Zhu et al. [32] introduces logic-in-memory (LiM) system that integrates 3D die stacked DRAM architecture with application specific data intensive applications.

All these works target a very specific set of applications like map reduce, graph processing, big data analysis etc. In addition, the work is primarily concentrated towards integrating PIM with conventional processing units. Hsieh et al. however approaches with GPU as the primary computing unit with data intensive code blocks offloaded to in-memory units. Our research targets both general purpose application and GPGPU integration to maximize utilization of PIM technology across a wide range of applications.

7 Conclusion

GPGPUs are very effective in executing programs with parallelism. Memory, however, also plays a very crucial role in deciding the system performance especially while executing applications with highly intensive and imbalanced workload. We integrate PIM (Processing in-Memory), with an effective technique to identify and offload memory intensive thread blocks, in the GPGPU system. Our technique dynamically identifies candidate blocks for offload using cache miss rate, block execution times and decides at runtime to actually offload based to resource availability at PIM. Our proposed solution improves GPGPU performance by 30% on an average across major general purpose workloads with up to twice performance increase in irregular memory intensive workloads. Block offloading reduces average block execution time and ensures uniform resource utilization and reduce energy consumption. Our approach also does not burden the programmer to make software changes for architectural compatibility.

Hence, we conclude that our approach with *Thread Block Offloading* to PIM presents a new dimension of solving memory bandwidth issues and efficiently utilizing PIM resources in GPGPU system.

8 Future Work

There are three major things we think should be continued in the future to make PIM block offloading more effective. First, to continue to further develop candidate block identification process by using static analysis using the compiler. Currently block identification uses previous block execution history and L1 cache miss rate to analyze and determine if a block would be running slow. However, getting recommendations from the compiler about all the memory intensive instructions in an application will further aid in identification process.

Second, handle divergence in block offloading. In our current work, thread blocks with divergence are restricted for offload to PIM even if the threads in that block are experiencing frequent cache misses. This is mainly because offloading divergent thread blocks adds complexity in tracking active threads and re-convergence program counters. Divergence is wide spread across many memory intensive applications and developing an technique capable of efficiently offloading divergent threads could potentially bring a lot of performance enhancements.

Third, dynamic block offload aggressiveness. Block offload aggressiveness controls the overall system performance: low aggressiveness will lead to underutilized PIM resources and high aggressiveness will make off chip links as a bottleneck also leading to insufficient utilization of PIM. Aggressiveness of block offload always needs to find a sweet spot to get the best possible performance result. Currently this is done statically but in future works this could be implemented in dynamic.

References

[1] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu and Michael Ignatowski. "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," *HPDC'14*, June 23–27, Vancouver, BC, Canada.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu and Kiyoung Choi. "PIM-Enabled Instructions: A Low-Overhead, Locality Aware Processing-in-Memory Architecture," ISCA'15, June 13–17, 2015, Portland, OR, USA

[3] Marko Scrbak, Mahzabeen Islam, Krishna M. Kavi, Mike Ignatowski, and Nuwan Jayasena. "Processing-in-Memory: Exploring the Design Space," *Springer International Publishing Switzerland*, 2015.

[4] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware," In *Eurographics* 2005, State of the Art Reports, August 2005, pp. 21-51.

[5] "Hybrid Memory Cube Specification 2.1," Hybrid Memory Cube Consortium, International Business Machines Corporation, Micron Technology, Inc.

[6] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu and Stephen W. Keckler. "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems."

[7] D. Patterson. “Why Latency Lags Bandwidth, and What it Means to Computing,” Keynote Address, Workshop on High Performance Embedded Computing, 2004.

[8] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. W. Jiang and Y. Solihin. “Scaling the Bandwidth Wall: Challenges in and Avenues for CMP scaling,” *36th International Symposium on Computer Architecture*, 2009.

[9] Paul Rosenfeld, Elliott Cooper-Balis, Todd Farrell, Dave Resnick, and Bruce Jacob. “Peering Over the Memory Wall: Design Space and Performance Analysis of the Hybrid Memory Cube.”

[10] L. Minas, “The Problem of Power Consumption in Servers,” 2009. [Online]. Available:http://software.intel.com/sites/default/files/m/d/4/1/d/8/power_consumption .pdf

[11] JEDEC, “Main Memory: DDR3 & DDR4 SDRAM,” http://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4sdram.

[12] www.micron.com/products/hybrid-memory-cube.

[13] G. Loh. “3D-stacked memory architectures for multi-core processors.” In 35th *International Symposium on Computer Architecture*, 2008.

[14] Y. Y. Pan and T. Zhang. “Improving VLIW Processor Performance using Three-Dimensional (3d) DRAM stacking,” In 20th *International Conference on Application-specific Systems, Architectures and Processors*, 2009.

[15] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee. “An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth,” In IEEE 16th *International Symposium on High Performance Computer Architecture*, 2010.

[16] K. Chen, S. Li, N. Muralimanohar, J.-H. Ahn, J. Brockman, and N. Jouppi, “CACTI-3DD: Architecture-level modeling for 3D Die [stacked DRAM main memory,” in *Design, Automation Test in Europe (DATE),* 2012, pp. 33–38.

[17] J. Jeddeloh and B. Keeth, “Hybrid Memory Cube New Dram Architecture Increases Density and Performance,” in *VLSI Technology (VLSIT), 2012 Symposium* on, June 2012, pp. 87–88.

[18] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron, “A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads,” IISWC '10 *Proceedings of the IEEE International Symposium on Workload Characterization* (IISWC'10).

[19] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt. “Analyzing CUDA Workloads Using a Detailed GPU Simulator.”

[20] GPGPUsim 3.x manual online. Available at http://gpgpu-sim.org/manual/index.php/Main_Page

[21] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing,” *ISCA’15*, June 13–17, 2015

[22] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. “GPUWattch: Enabling Energy Optimizations in GPGPUs,” *ISCA* ’13.

[23] D. Brooks, V. Tiwari and M. Martonosi. “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,” in proceedings of *ISCA*, 2000.

[24] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. “Memory-centric System Interconnect Design with Hybrid Memory Cubes,” 2013 *IEEE*.

[25] Tamara Schmitz. “The Rise of Serial Memory and the Future of DDR,” WP456 (v1.1) March 23, 2015. White Paper: Ultra Scale Devices.

[26] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, “Mapping Irregular Applications to DIVA, a PIM-based DataIntensive Architecture,” in Proceedings of *SC*, 1999.

[27] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas, “FlexRAM: Toward an Advanced Intelligent Memory System,” in Proceedings of *ICCD*, 1999.

[28] Seth H Pugsley, and Jeffrey Jestes. “NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads.”

[29] SAP, “In-Memory Computing: SAP HANA,” http://www.sap.com/ solutions/technology/in-memory-computing-platform.

[30] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *International Conference on Computer Design (ICCD)*, October 1999.

[31] Josep Torrellas. "FlexRAM: Toward an Advanced Intelligent Memory System A Retrospective Paper."

[32] Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry Pileggi, and Franz Franchetti, "A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing."

[33] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, and Hongjung Kim, "A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with e_ective microbump I/O test methods using 29nm process and TSV," in *ISSCC*, 2014.

[34] Ronald G. Dreslinski, David Fick, Bharan Giridhar, Gyouho Kim, and Sangwon Seo, "Centip3De: A 64-Core, 3D stacked nearthreshold system," *IEEE Micro*, 2013

[35] Dae Hyun Kim, Krit Athikulwongse, Michael Healy, and Mohammad Hossain, "3D-MAPS: 3D massively parallel processor with stacked memory," in *ISSCC*, 2012.

[36] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien, and Ravi Nair, "Data access optimization in a processing-in-memory system," in *CF*, 2015.

[37] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T. Kandemir, and Chita R. Das, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," *High Performance Computer Architecture (HPCA), IEEE International Symposium*, 2017

[38] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium*, 2015

[39] Lifeng Nai, RamyadHadidi, JaewoongSim, HyojongKim, PranithKumar, and HyesoonKim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," *High Performance Computer Architecture (HPCA), IEEE International Symposium*, 2017

[40] Black, B., Annavaram, M., Brekelbaum, N., and DeVale, "Die stacking (3D) microarchitecture," In: *Micro*, pp. 469-479. IEEE, 2006

[41] Hybrid Memory Cube Consortium, http://hybridmemorycube.org/

[42] Kogge P. M. "EXECUBE-A new architecture for scaleable MPPs," *International Conference on Parallel Processing*, *IEEE*, 1994

[43] Tor M. Aamodt, Wilson W.L. Fung, and Tayler H. Hetherington, GPGPU-Sim Manual.

[44] W. J. Dally and B. P. Towles, "Principles and Practices of Interconnection Networks," 2004.