Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs

Gwangsun Kim
Arm
gwangsun.kim@arm.com

Niladrish Chatterjee
NVIDIA
nchatterjee@nvidia.com

Mike O'Connor
NVIDIA and UT-Austin
moconnor@nvidia.com

Kevin Hsieh
Carnegie Mellon University
kevinhsieh@cmu.edu

ABSTRACT

3D-stacked memory devices with processing logic can help alleviate the memory bandwidth bottleneck in GPUs. However, in order for such Near-Data Processing (NDP) memory stacks to be used for different GPU architectures, it is desirable to standardize the NDP architecture. Our proposal enables this standardization by allowing data to be spread across multiple memory stacks as is the norm in high-performance systems without an MMU on the NDP stack. The keys to this architecture are the ability to move data between memory stacks as required for computation, and a *partitioned execution* mechanism that offloads memory-intensive application segments onto the NDP stack and decouples address translation from DRAM accesses. By enhancing this system with a smart offload selection mechanism that is cognizant of the compute capability of the NDP and cache locality on the host processor, system performance and energy are improved by up to 66.8% and 37.6%, respectively.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; **Multicore architectures**; • **Hardware → Emerging architectures**;

KEYWORDS

Near-data processing; 3D-stacked memory; GPU computing

1 INTRODUCTION

Memory bandwidth has long been one of the most critical bottlenecks [11] and 3D-stacked memory devices such as the Hybrid Memory Cube (HMC) [2] have been proposed as an alternative to conventional DDR and GDDR memory devices to overcome the bandwidth and energy limitations. However, while the HMC provides a large amount of DRAM bandwidth available through high-speed link interfaces, the GPU off-chip bandwidth is fundamentally limited and becomes a bottleneck for memory-intensive workloads. Especially as modern data center servers require several hundreds of GBs of memory capacity per node [42], a number of memory stacks are needed and the aggregate DRAM bandwidth available from them cannot be fully utilized by the GPU alone.

For example, to provide 128 GB memory, 16 HMCs are needed assuming 8 GB capacity per HMC. As each stack supports 320 GB/s of peak DRAM bandwidth, the aggregate bandwidth amounts to 5 TB/s. The HMC also provides sufficient off-chip bandwidth (480 GB/s based on current HMC specification [2]) to make the high DRAM bandwidth accessible through high-speed links. However, state-of-the-art GPUs [3, 46] only provide on the order of 1 TB/s or less off-chip bandwidth, resulting in ~4 TB/s bandwidth being unused.

One promising approach for improving the utilization of the memory bandwidth is near-data processing (NDP), which is to implement processing elements on the logic layer in the 3D-stacked memory device. The HMC also provides routing capability in the logic layer such that multiple HMCs can be interconnected through a memory network [29] and communicate with each other without consuming the scarce GPU memory bandwidth. As the number of processing elements in the logic layer and the bandwidth provided by the memory network scales with the number of memory stacks, the unused DRAM bandwidth can potentially be utilized through NDP to improve performance.

While NDP with a memory network is a promising direction to address the fundamental limitation of GPU off-chip bandwidth, it is very challenging to enable it in a *standardized* manner such that NDP can be done with future commodity memory devices. For decades, mainstream DRAM devices have remained as standardized, architecture-neutral components [4], and that has been the key reason for their economic scale. By imposing a standardized hardware interface for NDP, the memory stack can be designed independently from any specific GPU architecture (e.g., either NVIDIA or AMD). Unfortunately, all prior works on NDP either require an architecture-specific MMU/TLB on the logic layer [10, 19, 26, 34] or severely limit the memory access during NDP to a single memory stack [6, 17, 20, 37, 48], which often requires the programmer to manually place the data across multiple stacks [48]. Furthermore, it is impossible to completely bound memory access to a single memory stack for workloads with irregular access patterns (e.g., graph analysis) in general. Even for regular workloads, data placement can change during runtime in modern systems under *dynamic* memory management [14, 15, 24, 32]. Since any modern GPUs that support

SC17, November 12–17, 2017, Denver, CO, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126965

virtual memory requires an MMU,[1] it is a significant challenge to enable computation on memory stack without an MMU or TLB.

In this paper, we propose an architecture-neutral or standardizable NDP-enabled memory stack which can be exploited by different GPU architectures without any restriction on data placement. Our key contribution is a *partitioned execution* mechanism which decouples address translation (on the GPU) from DRAM access and register write-back (on the memory stack). With the partitioned execution, address translation is always performed on the GPU and the data movement is marshaled by the GPU. Thus, the memory-side core that we refer to as *NSU (Near-data processing SIMD Unit)* does not require an MMU. While removing the MMU from the memory stack reduces complexity and cost, the key contribution of this approach is that the memory stack can be built independently of the GPU's particular MMU implementation. Data movement between different memory stacks goes through a memory network, leveraging the unused HMC off-chip bandwidth without affecting the GPU traffic. Finally, to avoid cache coherence overheads, we avoid putting data caches on the NSU, but introduce NDP buffers instead.

However, the NSU can become a bottleneck with unconstrained offload as the NSUs provide relative low computational power compared to the SMs (Streaming Multiprocessors) in the GPU. In order to avoid the bottleneck and improve the utilization of the GPU SMs, we propose an algorithm to dynamically split the work between the GPU SMs and the NSUs. Furthermore, since cache-sensitive workloads can perform better when they are executed on the GPU to exploit the large on-chip caches, we propose a cache-aware offload decision algorithm to avoid performance loss for such workloads.

To the best of our knowledge, our approach is the first one to enable near-data processing for data distributed across multiple stacks while supporting virtual memory system without any architecture-specific component including MMU/TLB or data cache on the logic layer. Creating a standard framework for NDP can be beneficial as it creates the potential for commodity NDP-enabled memory stacks, leveraging economies of scale across the industry.

To summarize, the contributions of this work include the following:

- We propose a novel *partitioned execution* mechanism to enable near-data processing for data distributed across *multiple standardized* memory stacks. We eliminate the need for an architecture specific MMU/TLB or a data cache on the logic layer of memory stacks in enabling NDP.
- We show how our NDP architecture can reduce the waste of GPU off-chip bandwidth due to divergent memory accesses by not fetching untouched data to the GPU.
- We study the limited performance of a naive implementation of the proposed partitioned execution mechanism and improve performance through an algorithm to dynamically split the work between the GPU and NDP units and adjust the amount of computation offloaded.
- As offloading workloads with good cache locality to NDP units can result in performance degradation, we propose a dynamic, cache locality-aware offloading decision mechanism. The evaluation result shows that, with the dynamic offload ratio and cache locality-awareness, the performance is improved by up to 66.8% (17.9% on average) compared to the baseline.

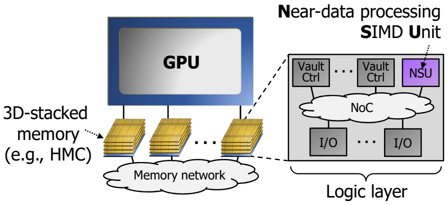


Figure 1: An overview of the proposed NDP architecture with a memory network and an NSU in the logic layer of each 3D-stacked memory.

2 PROPOSED ARCHITECTURE OVERVIEW

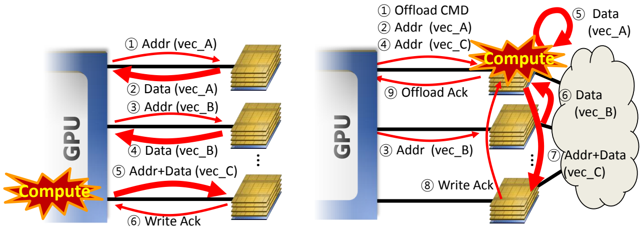
In this section, we provide an overview of our proposed standardizable NDP architecture for GPUs that does not restrict data placement. Our mechanism assumes simple, lightweight support from software to identify offload blocks at compile time (Section 3).

Figure 1 shows the overview of the proposed NDP architecture. We assume an HMC-like 3D-stacked memory, but other memory devices with abstract, packetized protocols such as Gen-Z [9] can be similarly extended to implement the NDP mechanism. In the logic die of the memory, in addition to vault controllers and I/O ports, there needs to be an NSU (Section 4.5) to execute offloaded instructions. One of our key contributions is a novel partitioned execution mechanism (Section 4.1) that splits the work between the GPU SM and the NSU without requiring a GPU-specific MMU in the memory stack. To support data communication between memory stacks, a memory network is created by leveraging the routing capability of the HMC. While communication between HMCs uses off-chip links, the off-chip bandwidth of the HMC is matched to the peak DRAM bandwidth and does not become a bottleneck [2]. Furthermore, since the communication between HMCs does not go through GPU links, GPU traffic is not affected. By leveraging NDP through a memory network, our proposed architecture can utilize the DRAM bandwidth that cannot be fully utilized by a GPU alone to improve performance for memory-intensive workloads.

Figure 2 shows an example of vector addition kernel execution on the baseline and our NDP-enabled GPU with our partitioned execution. Each thread reads two elements from two vectors (vector A and B), adds them, and writes the result into vector C. In the baseline (Figure 2(a)), the two vector data are fetched to the GPU (①-④), the computation is done on the GPU, and the result is written back to the memory (⑤-⑥). Most of the bandwidth is consumed by communicating data as indicated by the thick arrows in the figure since the overhead of command messages (with addresses) is amortized over multiple threads in a warp.

On the contrary, with our proposed NDP execution model shown in Figure 2(b), the data messages are communicated through a memory network, exploiting the HMC bandwidth that cannot be saturated by the GPU alone. The offloading is done in the unit of a warp (or wavefront) in our architecture. To initiate the offload, one of the memory stacks that has one of the data accessed is chosen as the NDP target, and its NSU is referred to as the *target NSU*. Then, an offload command message is sent to the target NSU (①),

[1] Recent GPUs have their own MMUs for accessing local graphics memory [33, 49].



(a) (b)

Figure 2: Vector addition (e.g., `C[tid]=A[tid]+B[tid]`) example with (a) baseline execution model and (b) the proposed NDP architecture. Thick arrows represent large messages.

and the GPU also immediately sends all memory read requests to the memories that have the data (② and ③) and provides write addresses for store instructions to the target NSU (④). For read requests, after DRAM is accessed, the data response is forwarded to the target NSU in the same HMC (⑤) or a different HMC (⑥) through the memory network. Then, the computation is done on the target NSU, and the write request for the result is sent to the appropriate memory (⑦) using the address provided by the GPU. After a write acknowledgment is received at the target NSU (⑧), an offload acknowledgment message is sent back to the GPU to notify that the offloaded execution is done (⑨). Instead of fetching the data to the GPU as in the baseline, read and write data are communicated through the memory network, reducing GPU off-chip bandwidth demand and mitigating the bandwidth bottleneck. The only message overhead we introduce is from the offload command and acknowledgment messages, but the overhead is amortized over multiple threads in a warp, which is the offload granularity. Note that the bandwidth consumption from sending memory addresses is the same for both the baseline and NDP system. Although we show a short, fine-grained offload example for simplicity, our NDP architecture enables both fine-grained and coarse-grained offloading.

In the following sections, we describe how the code for NDP is automatically generated for a target workload (Section 3) and the details of the proposed architecture to perform NDP (Section 4). Then, we evaluate and identify the limitation of a naive implementation of the proposed architecture (Section 6) and propose dynamic offloading decision mechanisms to achieve better performance (Section 7).

3 CODE GENERATION FOR NDP

3.1 Offload Block Identification

In this work, we refer to a segment of a given workload that can benefit from NDP as an *offload block*. An offload block can be manually identified by programmer directives or automatically identified by compiler analysis. In order to minimize programmer effort, we leverage an automatized approach proposed by prior work [26]. Their approach analyzes assembly code of a given workload and extracts offload blocks that can result in speedup by offloading. While they formulate a detailed equation for estimating the benefit and overhead, in order to leverage it during static analysis, we remove the cache hit rate and memory coalescing ratio terms that cannot be statically determined and use the following equation:

$$Score = GPUTrafficReduction - OffloadOverhead \qquad (1)$$

Although cache hit rate is not available during static analysis, we incorporate the workload's cache locality during runtime to improve offload decisions (Section 7). The score is used by the static analyzer to extract offload blocks from the workload during compile time. The term *GPUTrafficReduction* denotes the reduction in GPU's off-chip traffic by offloading the block to an NSU for each thread. For each load and store instruction in the block, *GPUTrafficReduction* is increased by the data size since offloading it will save the off-chip bandwidth by not transferring the data across the GPU's off-chip link. The GPU traffic for sending the accessed address is not considered in this equation since address needs to be sent off-chip regardless of whether the instruction is offloaded or not as shown in Figure 2.

However, since the offloaded computation may require some context information (i.e., register values) that resides in the GPU, the context needs to be transferred from the GPU to the NSU before offloading begins. Similarly, the GPU may require some register values generated within the offload block on the NSU to be sent back to the GPU at the end of the offloaded execution. The term *OffloadOverhead* in the equation reflects the amount of register transfer overhead to and from the GPU for the offloaded block and can be obtained from dependency between instructions through registers. However, instructions that calculate memory addresses for load and store instructions are not considered in the overhead calculation as they are executed on the GPU as described in (Section 4).

Similar to the prior approach [26], we avoid offload blocks that include scratchpad memory[2] instructions as workloads that effectively leverage the on-chip scratchpad memory can perform better on the GPU. In addition, synchronization among different threads in a thread block can be most efficiently done on the GPU and we do not include it in an offload block. An offload block needs to avoid spanning multiple basic blocks since control divergence during NDP is not desirable and can be better handled on the GPU [18, 28, 38].

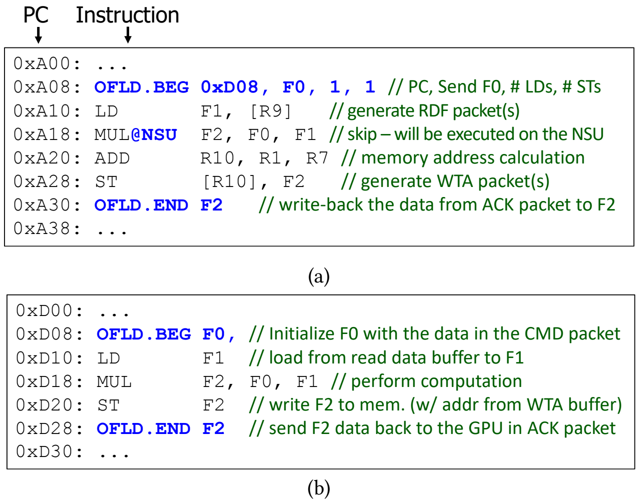
Furthermore, in order to overcome the GPU's memory divergence limitation with indirect memory accesses, we add any single indirect load instruction (i.e., memory access with address calculated from the value of another memory data) as an offload block. Since such a memory read is often divergent, offloading it to an NSU can significantly reduce GPU bandwidth waste and improve performance (Section 4.4).

While the focus of this work is on the architecture to enable standardized NDP, additional programmer input or sophisticated code analysis combined with compiler optimization can potentially further improve performance by resulting in more efficient or coarse-grained offload blocks. Although some of our evaluated workloads include relative small offload blocks (Section 5), our proposal is not limited to fine-grained offloading while we show that we improve performance and energy-efficiency even for fined-grained offloads.

3.2 Offload Block Code Generation

The information of offload blocks identified by the static analyzer needs to be embedded in the workload executable for the GPU as shown in Figure 3(a). In addition, for each identified offload blocks,

[2] The on-chip scratchpad memory is referred to as "shared memory" in CUDA and "local memory" in OpenCL.


PC
Instruction

```
0xA00: ...
0xA08: OFLD.BEG 0xD08, F0, 1, 1 // PC, Send F0, # LDs, # STs
0xA10: LD       F1, [R9]     // generate RDF packet(s)
0xA18: MUL@NSU  F2, F0, F1  // skip – will be executed on the NSU
0xA20: ADD      R10, R1, R7  // memory address calculation
0xA28: ST       [R10], F2    // generate WTA packet(s)
0xA30: OFLD.END F2     // write-back the data from ACK packet to F2
0xA38: ...
```

(a)

```
0xD00: ...
0xD08: OFLD.BEG F0,  // Initialize F0 with the data in the CMD packet
0xD10: LD       F1   // load from read data buffer to F1
0xD18: MUL      F2, F0, F1  // perform computation
0xD20: ST       F2   // write F2 to mem. (w/ addr from WTA buffer)
0xD28: OFLD.END F2   // send F2 data back to the GPU in ACK packet
0xD30: ...
```

(b)

Figure 3: An example code of an offload block for (a) GPU and (b) NSU. The embedded information for offload block is indicated with boldface text.

a corresponding code for the NSU needs to be generated with the NSU's ISA as shown in Figure 3(b) and appended to the original workload executable. Since the NSU provides a standardized ISA that can differ from that of the GPU, the instructions in the offloaded block need to be translated with the NSU's ISA. However, as the offload block only consists of simple load, store, and ALU instructions, the translation can be easily done by one-to-one mapping between the two ISAs, along with remapping register IDs.

Two new special instructions, OFLD.BEG and OFLD.END, are introduced to indicate the beginning and end of an offload block, respectively. They also list the registers that need to be transferred from the GPU to an NSU in the beginning and the registers that need to be transferred back to the GPU. In the GPU code, OFLD.BEG also specifies start PC address of the corresponding NSU code and the number of load and store instructions ("1, 1" in Figure 3(a)) such that the GPU SM can reserve buffers on the NSU.

Since memory addresses for all load and store instructions in an offload block are generated on the GPU before offload begins, any ALU instructions in the GPU code for memory address calculation are removed from the NDP code. For example, the ADD instruction for memory address calculation in Figure 3(a) is not included in the corresponding NSU code in Figure 3(b). On the other hand, the ALU instructions that are offloaded are marked by "@NSU" symbol and not executed on the GPU. The following section describes how the instructions are executed in detail.

4 NEAR-DATA PROCESSING MODEL

4.1 Partitioned Execution

A significant challenge in enabling NDP in a standardized manner without any restriction on data placement is address translation since a standardized memory stack cannot assume an architecture-specific MMU or TLB in its logic layer. In order to address the challenge, we propose a novel *partitioned execution* model. In this execution model, the work within an offload block is partitioned between the GPU and NSU such that address calculation, generation, coalescing, and sending memory access requests are done by the GPU while computation on the memory data is done on the NSU. In addition, in order to remove architecture-specific cache coherence from the logic layer, we do not assume a data cache in the logic layer but introduce NDP buffers instead as we describe below. In this work, we focus on the partitioned execution for throughput architectures, like GPUs, because these processors are the first systems for which stacked memories are being deployed [3, 46], and for CPUs, further studies need to be done to enable the partitioned execution for out-of-order cores.

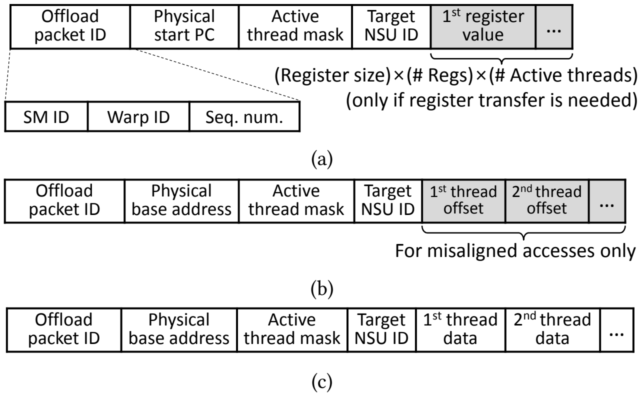


Figure 4: Offload packet format for (a) offload command packets, (b) read-and-forward (RDF) request and write address (WTA) packets, and (c) RDF response packets.

4.1.1 Partitioned Execution on the GPU

Begin offload (OFLD.BEG) instruction: Since the unit of control flow in GPU is a warp, the offloading from GPU SM and its execution on the NSU are also done in the unit of a warp. The GPU is responsible for generating different offload packets to initiate the offloaded execution on the NSU while transferring context (register) data and generating memory requests on behalf of the NSU. When OFLD.BEG instruction is executed on an SM at the beginning of an offload block, the SM generates an offload command packet with all information needed for initiation of offloaded execution on the NSU, including the register values that need to be transferred. Different fields of an offload command packet are shown in Figure 4(a). The first field of the packet is a unique offload packet ID, which is common for all types of packets introduced by our NDP mechanism. It is composed of the SM ID, warp ID, and the sequence number that is associated with each memory instruction. The offload command packet and the first set of packets that are generated from the first load or store instructions are given the sequence number of zero, and the following load or store instruction increments the sequence number by one.[3] The packet also specifies the *physical address* of the start PC, which is the address of the OFLD.BEG instruction in the code generated for the NSU (e.g., Figure 3(b)). We also assume that the pages for NSU code are mapped to physically contiguous memory region to remove any restriction due to page boundary. The active thread mask field indicates which threads in the warp are

[3] The number of bits for the sequence number in the packet format determines the maximum number of load and store instructions allowed in an offload block.

active. The shaded fields of the packet exist only if the `OFLD.BEG` instruction indicates that there are registers that need to be transferred. The target NSU specified is determined by the first memory instruction as explained in the description of memory instruction handling below.

In addition, with the number of load and store instructions within the block given by the `OFLD.BEG` instruction, the SM sends a reservation request to the GPU's on-chip NDP buffer manager (Section 4.3) to reserve *read data buffer* and *write address buffer*, which are needed to handle memory instructions, in the target NSU. The number of read data and write address buffer entries equals the number of load and store instructions within the block, respectively.

Memory instruction: The semantics of load and store instructions of partitioned execution is different from that of the normal execution mode. After addresses are generated and coalesced, instead of accessing data cache to fetch data to the register, read-and-forward (RDF) and write-address (WTA) packets (Figure 4(b)) are generated for load and store instructions, respectively. Compared to the baseline execution model, sending these packets does not necessarily increase energy or bandwidth consumption since the baseline needs to send memory read and write requests instead. The packet format is similar to that of the offload command packets up to the target NSU ID field, but instead of start PC, the physical cache line address that is accessed is provided. If the memory access pattern is divergent and a single load/store instruction touches multiple cache lines, one packet is generated for each memory address accessed in cache line granularity. In addition, the offset of the word accessed for each thread needs to be classified as either *aligned* or *misaligned*. A memory access is aligned if the offset of the word accessed by thread i ($0 \leq i < WarpWidth$) can be calculated as follows:

$$Addr_i = CacheLineBaseAddr + i \times WordSize$$

Otherwise, the memory access is classified as misaligned. For a misaligned access, the offset from the cache line base address for each thread is appended at the end the RDF or WTA packet as shown in Figure 4(b).

The HMC accessed by the first load or store instruction becomes the target NSU location. If multiple HMCs are accessed by the first memory instruction, the HMC with the most accesses from the instruction becomes the target. The target NSU, once determined, does not change during the execution of an offload block to minimize the overhead of communicating context. However, different *instances* of an offload block (e.g., an offload block executed multiple times within a loop) can be offloaded to different target NSUs depending on the data location. Figure 5 shows the impact of target NSU selection policy on memory traffic as the number of memory accesses within an offload block is varied. We assumed 8 HMCs and that the memory accesses are randomly mapped to the HMCs. Compared to the optimal policy that chooses the target based on all memory accesses in the offload block, our policy that only considers the first memory instruction increases the traffic by at most 15% only and the difference diminishes as the number of memory access increases. Since the optimal policy requires a huge amount of buffer to store all memory addresses generated, we choose the first HMC policy to avoid such overhead.

Figure 6(a) shows how RDF packets are communicated between the GPU and NSU. When an RDF packet is generated, it is first sent to the GPU on-chip caches and if it hits in the cache, an RDF response packet (Figure 4(c)) is generated with the cached data and sent to the target NSU to minimize DRAM access. Otherwise, the RDF packet is sent to the HMC's vault that the RDF address is mapped to and a response packet is generated with DRAM data and sent to the target NSU. Based on the active thread mask field of the RDF packet, only the data words that are actually accessed will be included in the response packet. For a write, a WTA packet is directly sent to the target NSU as shown in Figure 6(b), which will later use the packet to generate write requests for store instructions.

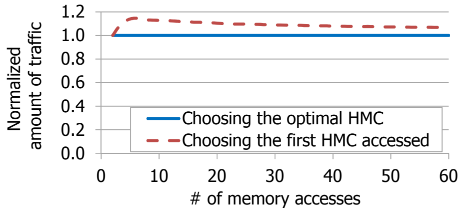
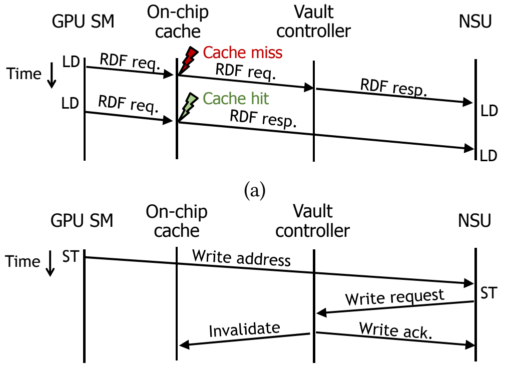


Figure 5: Impact of target NSU selection policy on off-chip memory traffic.



(a)

(b)

Figure 6: Timeline diagrams for how (a) load and (b) store instructions are executed in partitioned execution.

ALU instruction: Among all ALU instructions within an offload block, only those that calculate memory addresses are executed on the GPU (e.g., `ADD` instruction in Figure 3(a)) to generate and translate the address. Other ALU instructions will be executed on the NSU and they are *marked* at compile time as shown for the `MUL` instruction in Figure 3(a) and replaced with a NOP instruction on the GPU.

End offload (`OFLD.END`) instruction: This instruction signifies the end of an offload block. From this point, the execution of the warp will be blocked until the execution of the offloaded block on the NSU is finished and an acknowledgment from the target NSU is received at the GPU. However, the SM can context-switch to other warps to keep the SM busy.

NDP buffers in the GPU: There are two buffers – a *pending packet buffer* and a *ready packet buffer* in each GPU SM. If the target NSU is determined and the buffers on the NSU are granted by the buffer manager, the generated packet is queued in the ready packet buffer. Otherwise, it is queued in the pending packet buffer. A packet in the ready buffer can be sent to the destination HMC, but a packet in the pending buffer waits until the target NSU is determined and a buffer entry in the NSU is granted. When the state of the packet at the front of the pending queue is changed to ready state, it is sent to the target NSU.

Handling dynamic memory management: Although relatively infrequent, if a new page needs to be swapped in (e.g., between the host CPU and the GPU) while there is on-going offloaded execution, the writes for the new page need to be stalled until in-flight WTA messages to the same HMC are handled to guarantee correctness, while other HMC data can still be accessed. This can be done by keeping a counter of in-flight WTA packets for each HMC and waiting until the destination HMC has no in-flight WTA packets. The counter can be incremented for each WTA packet generated and decremented as the cache invalidation packet (Figure 6(b)) for a WTA is received at the GPU. The delay can be overlapped with the delay to fetch pages from an external interface such as NVLink or PCIe, which takes tens of microseconds [49]. In-flight RDF packets are not impacted as long as the write packets for the new page are sent through the same direct path to the HMC as the RDF packets, since the order will be preserved [2].

4.1.2 Partitioned Execution on the NSU

Begin offload (`OFLD.BEG`) instruction: When an offload command packet is received at an NSU and there is an empty warp slot available, a new warp is spawned on the NSU. The NSU's instruction fetch unit accesses the start PC given by the offload command through its physical instruction cache. The first instruction executed is the `OFLD.BEG` instruction, which can also specify the registers that need to be initialized with the data provided in the offload command packet. The offload packet ID for the current warp is initialized with the value provided in the command packet.

Memory instruction: For a load instruction, the NSU accesses the *read data buffer* with the current offload packet ID (Figure 4(a)). A read data buffer's entry consists of two fields – offload packet ID and the data for each thread. If the RDF packet has not arrived yet, the warp stalls until it arrives and the buffer entry is filled. If multiple RDF packets are generated from the GPU for a load instruction, their response packets are merged into a single entry in the read data buffer based on the packet's active thread mask. If all data have arrived for all active threads in the warp, they are loaded into the register specified by the load instruction. Similarly, for a store instruction, the *write address buffer* in the NSU is accessed with the current offload packet ID to obtain the physical memory addresses for the write. Then, as shown in Figure 6(b), the NSU generates write packets and sends them to the destination vault controllers (in the same or different HMCs), which will write the data and send acknowledgments back to the NSU. After a load or store instruction is executed, the current sequence number is incremented to identify the offload packet ID for the next memory instruction.

ALU instruction: ALU instructions in the offloaded code (e.g., `MUL` instruction in Figure 3(b)) can be executed normally with the data in the NSU's registers. Any ALU instructions that calculate memory addresses in the original code are removed from the NSU code during code translation.

End offload (`OFLD.END`) instruction: The warp waits until all in-flight DRAM writes are acknowledged, and then, the NSU generates an *offload acknowledgment packet* that consists of the current offload packet ID and register data that need to be transferred back to the GPU SM as a result of NDP. After the acknowledgment is sent to the GPU, the warp will be destroyed such that another offload command can be executed.

4.2 Cache Coherence

While we do not introduce data cache in the memory stack, the GPU's existing cache coherence needs to be preserved. In this work, we assume a relaxed memory consistency of CUDA and OpenCL programming models. With our mechanism, the NSU is guaranteed to load the most recent data through the RDF response packet since it is generated with cached data if there is a cache hit. In addition, when the offloaded execution is finished and the GPU thread resumes execution, it should obtain the most recent data written from the NSU. When the NSU performs a memory write, since the data are directly written to DRAM, the GPU cache can hold stale data which need to be invalidated. Thus, a cache invalidation message is generated from the vault controller to the GPU cache after each memory write is done. Based on our evaluation, the overhead from additional off-chip traffic is minimal (up to 1.42% and 0.38% on average for our evaluated workloads). Alternatively, the GPU can keep the store addresses generated within the offloaded block and invalidate them from the cache after the offloaded execution is finished, but we do not choose this approach to avoid the additional storage overhead.

4.3 Deadlock Prevention

Our mechanism also ensures deadlock-freedom for the NSU's buffers through credit-based buffer management. Deadlock can occur, for example, if the read data buffer is full of entries that will be accessed by later load instructions while there is an in-flight RDF response since it cannot be ejected from the network by the NSU. Thus, there is an NDP buffer management logic in the GPU that keeps the credit counts for the different buffers (i.e., offload command, read data, and write address buffers) in each HMC. The SM sends a buffer reservation request to the buffer manager on the GPU at the beginning of an offload block and the request is granted if there are sufficient credits. The NSU sends a credit to the buffer manager as buffer entries become available. The credit information can be piggybacked to other packets to minimize overhead.

4.4 Bandwidth Saving for Divergent Memory Access

For irregular workloads with divergent accesses, the GPU's off-chip bandwidth can be wasted by fetching all cache lines touched and then evicting them after little access to the data [39, 41]. In addition to reducing off-chip bandwidth consumption for regular workloads, our NDP mechanism can also reduce the waste of bandwidth due to divergent memory accesses by offloading each of them as a single-instruction offload block. Thus, for a memory access pattern that can be expressed as `x=B[A[i]]`, where `A` and `B` are arrays, the second load instruction that accesses array `B` is chosen as an offload block. Since the index for array `B` is calculated from memory data in array `A`, the indices accessed by the threads in the same warp can be very irregular and result in divergent accesses to different cache lines. If such an instruction is offloaded to an NSU, since the RDF response packets will only contain the data that are actually accessed by active threads, off-chip bandwidth will not be wasted to transfer data that will not be eventually accessed. Then, after all data are gathered in the read data buffer and loaded into the register on the NSU, the data will be sent back to the GPU in an offload acknowledgment packet. As a result, the warp in the GPU can fetch only the accessed data without fetching all unnecessary data in cache line granularity and polluting the cache. Among our evaluated workloads in Table 1, BFS and STCL include such offload blocks that consist of a single indirect load instruction.

4.5 NSU Design

The NSU can be implemented similar to the GPU SM but at a lower cost since it does not require several components of the SM. The NSU does not require data cache or scratchpad memory, and a small instruction cache and a register file can be sufficient since the instruction footprint for the offload blocks is small compared to that of the entire workloads. In addition, the NSU does not incur the MMU or TLB overhead, and the load/store unit is simplified since memory address generation and coalescing are done in the GPU and the NSU only accesses its local NDP buffers. Graphics-specific components such as texture unit with texture cache can also be removed from the NSU. Since the NSU executes memory-bound portions of the workload, it does not require high clock frequency and we assume it runs at half of the GPU SM frequency. We also study the impact of NSU clock frequency and show that running it at an even lower frequency can still provide most of the benefits (Section 7). The physical SIMD width of the NSU can also be made small while supporting larger or variable logical SIMD width through temporal SIMT [27].

5 EVALUATION METHODOLOGY

We modified GPGPU-sim [8] (version 3.2.0) to evaluate the performance of the proposed NDP mechanism. We model the pending and ready NDP buffers in the GPU and the different NDP buffers in the NSU, as well as the credit-based buffer management. The configuration of the system is given in Table 2 and we assume write-through policy for the GPU on-chip caches. The total number of SMs and bandwidth of the evaluated system are scaled down from the most recent GPUs for simulation feasibility, but we also present the impact of more powerful GPUs in Section 7.3. We focused on memory-intensive workloads listed in Table 1 from Rodinia [13], Parboil [44], NVIDIA CUDA SDK examples [1], and Polybench [22] for evaluation, as compute intensive workloads will result in no offloading by our optimization in Section 7 and the memory network and NSU can be power-gated. We performed static analysis of the workloads to automatically identify the offload blocks from the PTX assembly code and used it to evaluate the NDP mechanism. The last column of Table 1 lists the number of instructions in offload blocks translated for NSU (i.e., all unmarked ALU instructions that calculate memory addresses are removed). The number of registers transferred between the GPU and NSU is also statically determined and, for the evaluated workloads, the GPU transmitted (received) only 0.41 (0.47) registers per thread on average. For the memory network, we used 3D hypercube topology to interconnect 8 HMCs, using 3 links per HMC.[4] In order to properly evaluate the impact of data distributed across multiple HMCs, we used a random mapping of pages in 4 KB granularity. We modified GPUWattch [31] to model the system power including the NSU and memory network and we assumed off-chip link energy of 2 pJ/bit [36]. The power for the stacked memory is derived from the Rambus model [45], which models a DRAM device similar to that described in [30], and TSV models from [12]. From the models, the row activation energy is estimated to be 11.8 nJ for a 4 KB row [43] and the DRAM row buffer read energy is estimated as 4pJ/b. The wire energy for moving data on the GPU is estimated using values from [27], assuming a 20mm × 30mm GPU die.

Table 1: Evaluated workloads

Abbr.	Input problem	Description	Offload block size (# instr.)
BPROP	512K points	Back Propagation [13]	29, 23
BFS	1M nodes	Breadth-first search [13]	1,1,16
BICG	6K×6K	BiCGStab solver [22]	4,4
FWT	data: 2^{22}, kernel: 2^{17}	Fast Walsh Transform [1]	16,4
KMN	28k obj, 138 feat.	K-means [13, 40]	3
MiniFE	128×64×64	Finite element method [25]	3
SP	512 32K-vectors	Scalar product [1]	3
STN	512x512x64 grid	Stencil [44]	15
STCL	16k pts/blk, 1 blk	Streamcluster [13]	3,9,1,1
VADD	50M elements	Vector addition [1]	4

Table 2: System configuration

GPU	
Parameter	**Value**
# of SMs	64 SMs
# of HMCs	8
Off-chip link BW	20 GB/s for each direction of a link. Total 8 bidirectional links.
SM	1536 threads, 8 CTAs, 32768 registers, 48 KB scratchpad memory, warp width: 32
L1 inst. cache	4 KB, 4-way, 128 B line, MSHR: 2
L1 data cache	32 KB, 4-way, 128 B line, MSHR: 48
L2 cache	2 MB, 16-way, 128 B line, MSHR: 48
SM, Xbar, L2 clock	700, 1250, 700 MHz

HMC	
Parameter	**Value**
HMC organization	8 layers × 16 vaults, 16 banks/vault
HMC memory size	4 GB
Memory scheduler	FR-FCFS, Vault request queue size: 64

DRAM timing	tCK=1.50ns, tRP=9, tCCD=4, tRCD=9,
	tCL=9, tWR=12, tRAS=24

DRAM bandwidth	DDR3-1333H
Off-chip link BW	20GB/s in each direction for a link. Total 4 bidirectional links.

NDP-specific configuration	
Parameter	**Value**
NSU	350 MHz, 48 warps, warp width: 32 4 KB constant cache, 4 KB instruction cache
Buffers in GPU SM	8 B×300 entries for pending packet buffer 8 B×64 entries for ready packet buffer
Buffers in NSU	128 B×256 entries for read data buffer 128 B×256 entries for write address buffer 10 entries for offload command buffer

6 NAIVE IMPLEMENTATION RESULT

Figure 7 shows the performance of naively leveraging the proposed NDP mechanism. The NaiveNDP has a total of 72 SIMD units in the system – 64 SMs in the GPU and 8 NSUs, one per each HMC whereas the Baseline has only 64 SMs in the GPU. In order to evaluate the impact of the additional SIMD units, we also evaluate another baseline referred to as Baseline_MoreCore that has 8 additional SMs in the GPU. The result shows that compared to

[4] The HMC provides up to 4 links according to HMC specification 2.1 [2].

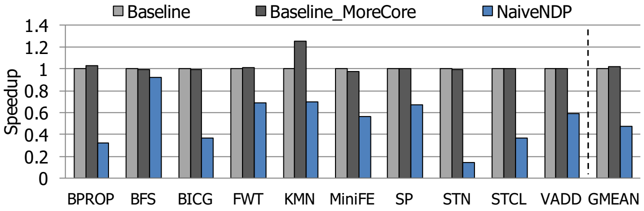


Figure 7: Performance of the naive NDP mechanism compared to different baselines.

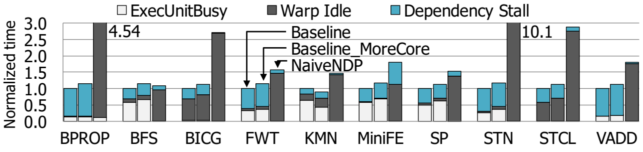


Figure 8: Breakdown of instruction no-issue cycles on the GPU. The numbers are normalized to the total no-issue cycles of the baseline.

the baseline, `Baseline_MoreCore` only resulted in less than 3% improvement for all workloads except for KMN which was improved by 25.7% due to the additional SMs' L1 data cache. However, the naive NDP mechanism resulted in performance degradation for all evaluated workloads – by up to 86% for STN and 52% on average. In order to analyze the performance degradation, we show a breakdown of instruction no-issue cycles on the GPU due to different reasons in Figure 8. The no-issue cycles are classified into three categories. "ExecUnitBusy" is when the execution unit was not available. "Dependency stall" is when an operand was not ready and this includes stalls due to cache or DRAM accesses. "Warp idle" is when a warp does not have a valid instruction to issue due to an empty instruction buffer, no active thread, or synchronization. For the NDP system, this category includes the cycles when warps are blocked while waiting for an acknowledgment packet from offloaded execution.

In the two baselines, since we focus on memory-intensive workloads, dependency stalls take a significant portion of no-issue cycles as memory bandwidth becomes a bottleneck, resulting in long memory access latency. For some workloads such as BFS and KMN, there is also significant stall time due to busy execution units, but overall, the warp idle cycles take a relatively small portion of no-issue cycles. However, with the naive NDP, the portion of warp idle cycles is significantly increased as many warps are stalled waiting for the offloaded blocks to finish on the NSUs, resulting in underutilization of the GPU and performance degradation. Thus, we propose improvements to the NDP mechanism to achieve speedups over the baseline in the following section.

7 IMPROVING PERFORMANCE THROUGH DYNAMIC OFFLOADING DECISION

7.1 Impact of Static Offload Ratio

As the NSU becomes the bottleneck for the naive NDP mechanism and the GPU is underutilized, we evaluated the NDP mechanism with *partial offloading*. With partial offloading, an offload block may or may not be offloaded depending on the output of offload decision logic. In order to study the impact of the amount of offloaded blocks on the performance, we varied the ratio of offload block instances from 0.2 to 1.0 with a step size of 0.2. Since the decision logic cannot be aware of the impact of offloading each instance of different blocks before executing it, we assumed the decision is randomly made to meet the given offload ratio. In this experiment, the ratio is static and does not change during workload runtime. The result in Figure 9 (`NDP(0.2-1.0)`) shows that performance can be improved by up to 63.6% for KMN and several other workloads benefit from 20-40% improvement depending on the offload ratio. BFS benefited from offloading divergent memory accesses and resulted in 31% speedup for offload ratio of 0.4. However, there is only degradation for BPROP, BICG, STN, and STCL. For BPROP, the performance was further degraded as offload ratio was increased as it has a small, constant data structure of 68 bytes regardless of input problem size that is accessed within all offload blocks. In the baseline, accesses to this data structure mostly result in cache hits, but with the NDP mechanism, the data are always transferred off-chip from the GPU to the NSU after RDF requests hit in the cache and the off-chip bandwidth of the GPU becomes the bottleneck. Such a workload can benefit from adding a small read-only cache to each NSU with minimal cost. The performance of BICG did not improve with the evaluated static offloading but it was due to the large granularity we used to change the offload ratio and there was performance improvement with smaller offload ratio. For example, the offload ratio of 0.15 resulted in an 11.5% speedup. For STN, the baseline showed a moderate cache hit rate of 45% in the L2 cache for read accesses, and thus, the NDP resulted in performance degradation as it increased DRAM accesses. The performance of STCL was degraded for the same reason. For other workloads, as offload ratio is increased, the performance improves until the NSU becomes a bottleneck, and then the performance degrades as more blocks are offloaded. The evaluated workloads resulted in different optimal offload ratios as they have different amount of memory accesses, memory divergence, and the amount of computation within the offload blocks. There was no single static offload ratio that resulted in the best performance for all workloads. Thus, we propose dynamically determining the offload ratio as described in the following subsections.

7.2 Dynamic Offload Ratio

In this section, we propose an algorithm to dynamically determine a near-optimal offload ratio based on hill climbing method in Algorithm 1. In order to minimize the overhead of executing the algorithm, we propose an epoch-based approach where the offload ratio is computed once at the end of each long epoch and used throughout the next epoch. Since the algorithm is simple, it can be executed within a short time and during its execution, offload ratio from the previous epoch can be used with little impact on performance, assuming sufficiently large epoch length.

The algorithm measures the instruction throughput with a given offload ratio during each epoch and compares the throughput with that of the previous epoch. If the throughput is increased, the offload ratio is further moved in the same direction (i.e., keep increasing or decreasing) by the current step size. On the contrary, if the throughput is decreased, the offload ratio is varied in the opposite

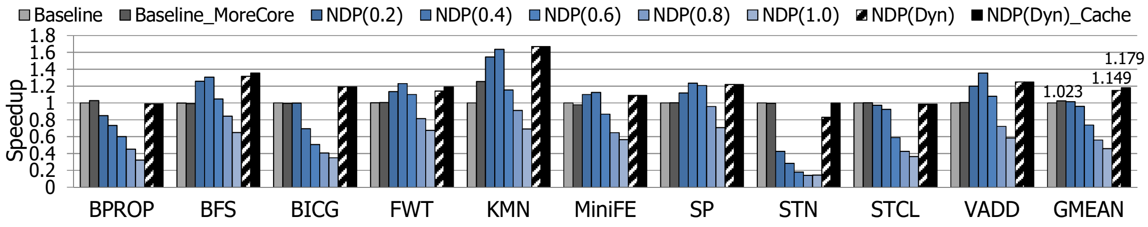


Figure 9: Performance of the NDP as offloading ratio is statically varied, and NDP with dynamic offloading decision.

Algorithm 1 Dynamic offload ratio decision algorithm based on hill climbing method

```
init: $Step_{cur} \leftarrow Step_{max}$, $Ratio_{cur} \leftarrow Ratio_{init}$

At the end of each epoch except for the first:
if $cur\_avg\_ipc < prev\_avg\_ipc$ then
    $dir \leftarrow dir \times (-1)$        ▷ Reverse direction if getting worse
    $dir\_change\_history$.push_back ($true$)
else
    $dir\_change\_history$.push_back ($false$)
end if

if $dir\_change\_history$.size > $Window\_size$ then
    pop front from $dir\_change\_history$
end if

$N\_changes \leftarrow$ # of $true$ in $dir\_change\_history$
if $N\_change > Window\_size/2$ AND $Step_{min} < Step_{cur}$ then
    $Step_{cur} \leftarrow Step_{cur} - Step_{unit}$
else if $Step_{cur} < Step_{max}$ then
    $Step_{cur} \leftarrow Step_{cur} + Step_{unit}$
end if

if $Step_{unit} \leq Ratio_{cur} \leq 1.0 - Step_{unit}$ then
    $Ratio_{cur} \leftarrow Ratio_{cur} + dir \times Step_{unit}$
end if
```

direction (i.e., start increasing if it was decreasing or vice versa) by the current step size. In order to accurately measure the impact of the offload ratio, the instruction throughput only considers the instructions within the offload block regardless of whether it was offloaded or not. However, since the algorithm keeps trying different offload ratios, it can result in oscillation instead of maintaining the optimal ratio after it is found. On the other hand, the characteristics of a workload can change during execution and it can be beneficial to keep trying different offload ratios. Thus, our proposed algorithm uses an *adaptive* step size based on recent history of changes in the direction of movement. If the offload ratio keeps moving in the same direction, using a larger step can result in reaching the optimal offload ratio faster (or within a small number of epochs). On the other hand, if the offload ratio continually reverses the direction of movement, it indicates that the optimal offload ratio is close to current ratio, and thus, using a smaller step will result in more closely approaching the optimal ratio while reducing the impact of oscillation. Thus, the algorithm keeps a history of the changes in the direction of movement (either changed or not changed) and if there were more changes in the direction than moving in the same direction, a smaller step is used. Otherwise, a larger step size is used to vary the offload ratio. We impose a lower bound and an upper bound on the step size to avoid becoming susceptible to small noise due to too small step sizes or missing the optimal ratio due to too large step sizes. In this work, we assume an epoch length of 30,000 cycles, an initial offload ratio of 0.1, an initial step size of 0.15, the granularity of step size change of 0.05, maximum and minimum step sizes of 0.05 and 0.15, respectively, and history window size of 4. Since the algorithm is simple, it can be either implemented in hardware or executed on the GPU's embedded CPU [47]. Once the offload ratio is determined, it is used to determine whether each block instance will be offloaded, similar to the static offload ratio.

The result of the proposed dynamic offload ratio decision algorithm is shown in Figure 9 as NDP(Dyn). For most workloads, the algorithm resulted in performance close to that of the best offload ratio, resulting in a speedup of up to 66.8% for KMN, and 14.9% on average. For VADD, due to the oscillation that can still occur, the dynamic ratio resulted in 9% lower performance than that of the best static ratio, but still provided 24.9% speedup over the baseline. On the other hand, the oscillation degraded the performance of STN by 17% compared to the baseline. As this workload exhibits good cache locality as described in Section 7.1, the optimal offload ratio is 0 and not offloading any block gives the best performance. However, the dynamic offload ratio decision algorithm continually tries non-zero offload ratios although it is varied in a small step. In general, different offload blocks can have different cache locality, and the cache-sensitive offload blocks need to be suppressed from being offloaded while other blocks still benefit from offloading. Thus, we propose incorporating each offload block's cache locality in the offload decision in the next subsection.

7.3 Cache Locality-Aware Offload Decision

Workloads with good cache locality can execute more efficiently on the GPU than on the NSU due to the high bandwidth and low energy of the GPU's on-chip caches. While static analysis at compile time can often identify the memory access pattern and may be able to infer cache locality information, it is limited to regular workloads, and the cache behavior is unpredictable for irregular workloads at compile time. Thus, the cache behavior needs to be measured at runtime for high accuracy. While prior work [6] proposed cache locality-aware offloading decision, their NDP mechanism was limited to only accessing a single cache line during offloaded execution and thus, is not applicable to our approach.

In this work, we propose to measure the cache behavior of the offload blocks and use the information to suppress offloading the

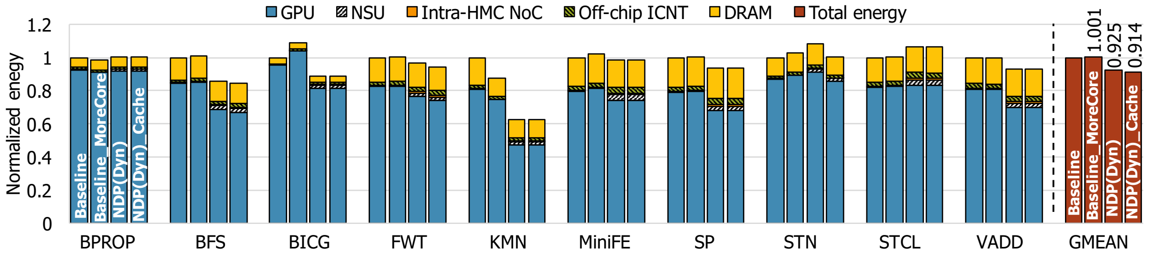


Figure 10: Normalized energy for different baselines and NDP mechanisms.

block if the overhead outweighs the benefit. For each offload block, the number of RDF packets generated and the number of cache hits for the RDF packets are accumulated to obtain the average number of RDF packets generated (*AvgNumCacheLines*) and the average miss rate for the RDF packets (*AvgCacheMissRate*). Then, the benefit of the offload block is calculated by the following equation.

$$\begin{aligned} Benefit = & \lceil AvgNumCacheLines \times AvgCacheMissRate \rceil \\ & \times CacheLineSize \times SIMDWidth \\ & + NumStoreInsts \times WordSize \times SIMDWidth \end{aligned}$$

The first and second term estimate the benefit from offloading load and store instructions within the block, respectively. In the first term, the average number of cache lines accessed is multiplied by the miss rate to estimate the number of cache lines that need to be fetched from DRAM if this offload block were to be executed on the GPU. This gives the benefit from offloading the load instructions as offloading them will result in GPU traffic reduction. The ceiling function is needed since data fetch is done in cache line granularity. For store instructions, since we assume a write-through cache, the amount of data that need to be transferred off-chip equals the number of data words written. The value of *NumStoreInsts* is specified by the `OFLD.BEG` instruction as described in Section 3.2. The *Benefit* calculated is substituted into *GPUTrafficReduction* in Equation 1 to obtain the score of an offload block and if the score is not positive, the block is not offloaded. Otherwise, the block can be offloaded based on current offload ratio that is dynamically determined by Algorithm 1.

Performance improvements by the cache locality-aware offload decision are shown in Figure 9 as the last column for each workload (`NDP(Dyn)_Cache`). The performance for `STN` is improved compared with `NDP(Dyn)` as its offload blocks are suppressed from being offloaded based on the cache locality. The overall speedup compared to the baseline is increased from 14.9% to 17.9% while other workloads were nearly unaffected. We also studied the impact of a more powerful GPU. When the number of compute units in the GPU is doubled for all configurations, the proposed offloading mechanism resulted in an 11.6% speedup over baseline on average. Even with more compute units and caches in the GPU, the off-chip bandwidth can still become a bottleneck and we overcome the limitation through memory network-based NDP.

7.4 Energy Reduction

Figure 10 compares the energy consumption of the proposed NDP model with that of the baseline execution model. Compared to the baseline, baseline with more SMs resulted in nearly the same energy as the baseline as a slight reduction in runtime was offset by the increased power consumption by additional SMs. On the contrary, the proposed NDP mechanism reduced energy by up to 37.6% for `KMN` and by 7.5% on average with `NDP(Dyn)`. `NDP(Dyn)_Cache` resulted in a higher reduction of 8.6% with cache locality awareness. Note that this energy reduction takes into account the additional HMC links for the memory network and the increase in data traffic due to NDP.

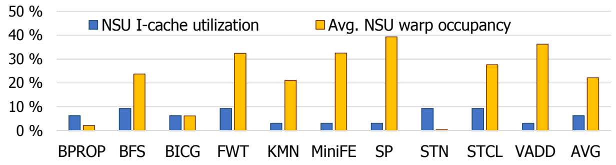


Figure 11: I-cache utilization and warp occupancy of NSU.

7.5 Hardware Overhead

Our NDP mechanism introduces little hardware overhead in the GPU. For the configuration we assume in our evaluation (Table 2), each SM requires 2.84 KB storage for the pending and ready packet buffers which is significantly smaller than existing per-SM on-chip storage including L1 data and instruction caches, scratchpad memory, constant cache, and texture cache, as well as the L2 cache in modern GPUs. For the configuration we assume, the storage overhead is only 1.8% of total on-chip storage on the GPU.

In addition, Figure 11 shows the utilization of instruction cache and warp occupancy of the NSU during runtime. As we only offload memory-intensive segments of the workload, the instruction cache shows very low utilization of 23.7% out of 4 KB per NSU. In addition, the average SIMD thread occupancy is at most 39.3% (for SP) and 22.1% on average out of 48 available SIMD hardware threads. Thus, the NSU can be implemented at low cost as well.

7.6 Performance Sensitivity to NSU Frequency

Since the goal of our NDP mechanism is to offload memory-intensive segments of workloads, the NSU can be mainly memory-bound rather than being compute-bound. In addition, since one of the considerations of NDP is the thermal impact of introducing a compute unit in the memory stack, it is desirable to run the NSU at a low frequency. Thus, we studied the impact of reducing the clock frequency of the NSU by half. Compared to the 350 MHz NSU result shown in Figure 9, 175 MHz NSU still achieved a speedup of up to 67.7% (for `KMN`) and 14.1% on average when the dynamic offload ratio decision and cache-locality awareness were used. Because of the low performance requirement of the NSU, it can be fabricated in a cheaper, older technology compared to DRAM and GPU, and run at a low frequency.

8 RELATED WORK

There have been many researches that proposed processing-in-memory through single-die integration of logic and DRAM in the late 90s to early 2000s, but they were limited by high manufacturing cost and unconventional programming models. Several recent works [5, 23, 37] proposed NDP for specific target workloads such as MapReduce, key-value store, or graph processing, but we focus on standardizable NDP architecture that is not limited to any framework. While several prior works investigated the design of general-purpose NDP systems, they either restricted data access during NDP to a single memory device or introduced an architecture-specific MMU or TLB in the logic layer of the memory.

8.1 Prior Work with Data Access Restriction

Terasys [21] was one of the early designs of processing-in-memory (PIM), where SIMD processing was done in memory devices. It supported communication among PIM chips to perform reduction operation, but unstructured data had to be sent through the host processor. PEI (PIM-Enabled Instructions) [6] proposed special instructions that can perform computation on the logic layer of the HMC to accelerate data-intensive workloads. However, one of its limitations is that only a single cache line can be accessed during the offloaded computation while we overcome such a limitation. HRL [20] combined both fine-grained and coarse-grained blocks for NDP to achieve high performance as well as high power-efficiency but did not support virtual memory. They propose NDP within special, non-cacheable region to avoid the virtual memory translation and cache coherence protocol. TOP-PIM [48] studied the impact of heterogeneous processing units with CPU and GPU cores on the logic layer of 3D-stacked memories. They presented simulation methodology for performance and power evaluation of NDP architecture as well as characterization of workloads in the context of NDP. They assumed that programmer manually places the data across multiple stacks while the NDP logic can only access the data in the same memory stack. NDA [17] proposed stacking a logic layer on top of a DRAM device to enable NDP while providing a standard DDR memory interface to the processor. They assumed CGRA (Coarse-grain Reconfigurable Accelerator)-based processing element for NDP. Although they did not introduce an architecture-specific component in the logic layer, only a single memory device can be accessed during NDP and the data need to be shuffled while the ownership of the DRAM is switched between the processor and the CGRAs. Chameleon [7] proposed a near-DRAM accelerator that can be integrated to data buffers of conventional LRDIMM, without requiring 2.5D or 3D integration. Pattnaik et al. [35] proposed scheduling techniques for an NDP-enabled GPU system running multiple concurrent kernels. They presented an execution time predictor that guides the scheduler in determining where each kernel will be scheduled between the main GPU and the GPU-PIM in the memory stack. While they evaluated the performance with multiple memory stacks, each GPU-PIM was assumed to only access the data within a single memory device.

8.2 Prior Work with Architecture-specific MMU/TLB

DIVA [16] was another early work that demonstrated the benefits of PIM. Communication between PIM devices was supported by a dedicated network while address translation was done within the PIM device. Active Memory Cube [34] focused on NDP through vector processing units in the 3D-stacked memory. They introduced an MMU on the logic layer and their vector units only processed data within the same memory stack. A recent prior work [26] proposed automatically identifying offload blocks within GPU workloads and a mechanism to dynamically determine the memory mapping of data across multiple stacks in a programmer-transparent manner. However, they also assumed an architecture-specific TLB on the logic layer. In addition, while modern GPUs support dynamic memory management between CPU and GPU, they assumed that memory mapping does not change during execution to avoid TLB shootdown for the memory-side TLB. Gao et al. [19] proposed hardware/software architectures to realize a practical NDP system and presented design space exploration. They assumed architecture-specific TLB on the logic layer to enable memory access across multiple stacks for NDP.

9 CONCLUSION

We proposed an architecture-neutral mechanism to perform NDP in a standardized manner while removing any restriction on data placement across multiple memory stacks. Our proposed architecture also overcomes the limitations of prior work as we do not require the programmer to manually specify data placement. We designed a partitioned execution mechanism where the GPU performs address generation and translation while coordinating data movement between memory stacks, thereby avoiding architecture-specific address translation on the memory stacks. We evaluated the performance of a naive implementation of the NDP mechanism and proposed an algorithm to dynamically adjust the offload ratio to achieve higher performance. Furthermore, we also proposed a cache locality-aware offload decision mechanism to prevent performance degradation for workloads with good cache locality. Our evaluation results show that the proposed mechanism achieved a speedup of up to 66.8% (17.9% on average) and an energy reduction of up to 37.6% (8.6% on average) while incurring low hardware overhead.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the U.S. Department of Energy.

REFERENCES

[1] 2011. CUDA C/C++ SDK code samples. NVIDIA. (2011).
[2] 2014. Hybrid Memory Cube Specification 2.1. Hybrid Memory Cube Consortium. (2014).
[3] 2016. NVIDIA Tesla P100. (2016). NVIDIA white paper.
[4] 2017. JEDEC Solid State Technology Association. (2017). http://jedec.org.
[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[6] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15).*

[7] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16).*

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09).*

[9] Brad Benton. 2017. CCIX, GEN-Z, Open CAPI: OVERVIEW & COMPARISON. 13th Annual OpenFabrics Alliance Workshop. (Mar 2017).

[10] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. 2017. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE Computer Architecture Letters* 16, 1 (Jan 2017).

[11] Doug Burger, James R. Goodman, and Alain Kägi. 1996. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96).*

[12] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens. 2013. System and circuit level power modeling of energy-efficient 3D-stacked wide I/O DRAMs. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE).*

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09).*

[14] Hanjin Chu. 2013. AMD Heterogeneous Uniform Memory Access. AMD. (2013).

[15] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13).*

[16] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-memory Chip. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02).*

[17] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15).*

[18] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '07).*

[19] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT '15).*

[20] M. Gao and C. Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *the IEEE International Symposium on High Performance Computer Architecture (HPCA '16).*

[21] M. Gokhale, B. Holmes, and K. Iobst. 1995. Processing in memory: the Terasys massively parallel PIM array. *Computer* 28, 4 (Apr 1995), 23–31.

[22] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar).*

[23] Anthony Gutierrez, Michael Cieslak, Bharan Giridhar, Ronald G. Dreslinski, Luis Ceze, and Trevor Mudge. 2014. Integrated 3D-stacked Server Designs for Increasing Physical Density of Key-value Stores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14).*

[24] Mark Harris. 2013. Unified Memory in CUDA 6. GTC On-Demand, NVIDIA. (2013).

[25] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).

[26] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16).*

[27] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *Micro, IEEE* 31, 5 (Sept 2011), 7–17.

[28] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15).*

[29] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. 2013. Memory-centric System Interconnect Design with Hybrid Memory Cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13).*

[30] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. 2014. A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC '14).*

[31] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13).*

[32] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'15).*

[33] Wen mei W. Hwu. 2012. *GPU Computing GEMS Jade Edition.* Morgan Kaufmann.

[34] R. Nair, S.F. Antao, C. Bertolli, P. Bose, J.R. Brunheroto, T. Chen, C. Cher, C.H.A. Costa, J. Doi, C. Evangelinos, B.M. Fleischer, T.W. Fox, D.S. Gallo, L. Grinberg, J.A. Gunnels, A.C. Jacob, P. Jacob, H.M. Jacobson, T. Karkhanis, C. Kim, J.H. Moreno, J.K. O'Brien, M. Ohmacht, Y. Park, D.A. Prener, B.S. Rosenburg, K.D. Ryu, O. Sallenave, M.J. Serrano, P.D.M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* (2015).

[35] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16).*

[36] J. Poulton, R. Palmer, A. M. Fuller, T. Greer, J. Eyles, W. J. Dally, and M. Horowitz. 2007. A 14-mW 6.25-Gb/s Transceiver in 90-nm CMOS. *Solid-State Circuits, IEEE Journal of* 42, 12 (2007).

[37] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '14).*

[38] Minsoo Rhu and Mattan Erez. 2013. The Dual-path Execution Model for Efficient GPU Control Flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '13).*

[39] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '13).*

[40] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '12).*

[41] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '13).*

[42] Mark Shaw, Martin Goldstein, and Mark A. Shaw. 2016. Open CloudServer OCS Blade Specification Version 2.1. Open Compute Project. (Feb 2016).

[43] Young Hoon Son, O. Seongil, Hyunggyun Yang, Daejin Jung, Jung Ho Ahn, John Kim, Jangwoo Kim, and Jae W. Lee. 2014. Microbank: Architecting Through-silicon Interposer-based Main Memory Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14).*

[44] J.A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. IMPACT Technical Report, Center for Reliable and High-Performance Computing. (2012).

[45] Thomas Vogelsang. 2010. Understanding the Energy Consumption of Dynamic Random Access Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10).*

[46] S. Wasson. 2015. AMD Radeon R9 Fury X Graphics Card Reviewed. (2015). http://techreport.com/review/28513/amd-radeon-r9-fury-x-graphics-card-reviewed.

[47] Joe Xie. 2016. NVIDIA RISC-V Evaluation Story. In *Proceedings of the 4th RISC-V workshop.*

[48] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14).*

[49] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16).*