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

Gwangsun Kim
Arm
gwangsun.kim@arm.com

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

Niladrish Chatterjee
NVIDIA
nchatterjee@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 virtual memory requires an MMU, 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 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.

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.
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.

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), 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:

$$\text{Score} = \frac{\text{GPU Traffic Reduction}}{\text{Offload Overhead}}$$  

(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 GPU Traffic Reduction 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, GPU Traffic Reduction 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 Offload Overhead 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 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, a corresponding code for the NSU needs to be generated from the GPU code that includes the register values that need to be transferred. Different offload blocks, the SM generates an offload command packet with all information needed for initiation of offloaded execution on the NSU, the SM generates an offload command packet with all information needed for initiation of offloaded execution on the NSU. The offload command packet contains the offload block ID, physical base address, active thread mask, target NSU ID, and first register value.

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.

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 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.

![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.](image-url)

![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.](image-url)
and the following load or store instruction increments the sequence number by one. 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 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 \( i (0 \leq i < \text{Warp Width}) \) can be calculated as follows:

\[
\text{Addr}_i = \text{CacheLineBaseAddr} + i \times \text{WordSize}
\]

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 in the offload block 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.

**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 NSF is determined and the buffers on the NSF 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 NSF is determined and a buffer entry in the NSF 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 NSF.

**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 RDP packets, since the order will be preserved [2].

### 4.1.2 Partitioned Execution on the NSF

**Begin offload (OFLD.BEG) instruction:** When an offload command is received at an NSF and there is an empty warp slot available, a new warp is spawned on the NSF. The NSF’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 NSF 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 NSF 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 NSF 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 NSF. 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 NSF’s registers. Any ALU instructions that calculate memory addresses in the original code are removed from the NSF code during code translation.

**End offload (OFLD.END) instruction:** The warp waits until all in-flight DRAM writes are acknowledged, and then, the NSF 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 NSF 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 NSF. When the NSF 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 NSF’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 NSF. 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 NSF sends a credit to the buffer manager as buffer entries become available. The credit information can be piggy-backed 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. 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 GPUWatch [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 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](image)

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Input problem</th>
<th>Description</th>
<th>Offload block size (# instr.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>6k×6k K</td>
<td>BiCGStab solver [22]</td>
<td>4, 4, 4</td>
</tr>
<tr>
<td>FWT</td>
<td>data: 2D, kernel: 2D</td>
<td>Fast Walsh Transform [1]</td>
<td>16, 4, 1</td>
</tr>
<tr>
<td>KMIN</td>
<td>2k obj, 138 feat.</td>
<td>K-means [13, 40]</td>
<td>3, 3</td>
</tr>
<tr>
<td>MiniK</td>
<td>128x64x64</td>
<td>Scalare method [19]</td>
<td>3, 3</td>
</tr>
<tr>
<td>SP</td>
<td>512 32K vectors</td>
<td>Scalar product [1]</td>
<td>3</td>
</tr>
<tr>
<td>STN</td>
<td>512x512x64 grid</td>
<td>Stencil [44]</td>
<td>3, 3, 3</td>
</tr>
<tr>
<td>STCL</td>
<td>16k pts/bk, 1 blk</td>
<td>Streamcluster [13]</td>
<td>3, 3, 3</td>
</tr>
<tr>
<td>VADD</td>
<td>50M elements</td>
<td>Vector addition [1]</td>
<td>4, 4, 4</td>
</tr>
</tbody>
</table>

![Table 2: System configuration](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of HMCs</td>
<td>8</td>
</tr>
<tr>
<td># of SMs</td>
<td>9</td>
</tr>
<tr>
<td>Off-chip link BW</td>
<td>20 GB/s for each direction of a link, Total 8 bidirectional links.</td>
</tr>
<tr>
<td>SM</td>
<td>1536 threads, 8 CTA s, 32/67 registers, 48 KB scratchpad memory, warp width: 32</td>
</tr>
<tr>
<td>L1 inst. cache</td>
<td>4 KB, 4-way, 128 B line, MSHR: 2</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>32 KB, 4-way, 128 B line, MSHR: 48</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2 MB, 16-way, 128 B line, MSHR: 48</td>
</tr>
<tr>
<td>L2 cache</td>
<td>700, 1250, 700 MHz</td>
</tr>
<tr>
<td>L2 cache</td>
<td>700, 1250, 700 MHz</td>
</tr>
<tr>
<td>HMC organization</td>
<td>8 layers × 16 vaults, 16 banks/vault</td>
</tr>
<tr>
<td>HMC memory size</td>
<td>4 GB</td>
</tr>
<tr>
<td>Memory scheduler</td>
<td>FR-FCFS, Vault request queue size: 64</td>
</tr>
<tr>
<td>DRAM timing</td>
<td>tCL=3,1 MHz, BPP=9, tCCD=4, tWR=12, tRAS=24</td>
</tr>
<tr>
<td>DRAM bandwidth</td>
<td>1083×1333 H</td>
</tr>
<tr>
<td>Off-chip link BW</td>
<td>200 GB/s in each direction of a link, Total 4 bidirectional links.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers in GPU SM</td>
<td>8 80x300 entries for pending packet buffer, 8 80x64 entries for ready packet buffer</td>
</tr>
<tr>
<td>Buffers in NSU</td>
<td>128 B×256 entries for read data buffer, 10 entries for offload command buffer</td>
</tr>
</tbody>
</table>

The HMC provides up to 4 links according to HMC specification 2.1 [2].
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 only has 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 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
Algorithm 1 Dynamic offload ratio decision algorithm based on hill climbing method

\[
\text{init: Step}_\text{cur} \leftarrow \text{Step}_\text{max}, \text{Ratio}_\text{cur} \leftarrow \text{Ratio}_{\text{init}}
\]

At the end of each epoch except for the first:

if \(\text{cur}_\text{avg}_\text{ipc} < \text{pre}_\text{avg}_\text{ipc}\) then

\[
\text{dir} \leftarrow \text{dir} \times (-1) \quad \text{Reverse direction if getting worse}
\]

\[
\text{dir}_\text{change}_\text{history}.\text{push}_\text{back} \text{ (true)}
\]

else

\[
\text{dir}_\text{change}_\text{history}.\text{push}_\text{back} \text{ (false)}
\]

end if

if \(\text{dir}_\text{change}_\text{history}.\text{size} > \text{Window}_\text{size}\) then

pop front from \(\text{dir}_\text{change}_\text{history}\)

end if

\[
N_{\text{changes}} \leftarrow \# \text{of true in } \text{dir}_\text{change}_\text{history}
\]

if \(N_{\text{change}} > \text{Window}_\text{size}/2 \AND \text{Step}_\text{min} < \text{Step}_\text{cur}\) then

\[
\text{Step}_\text{cur} \leftarrow \text{Step}_\text{cur} - \text{Step}_\text{unit}
\]

else if \(\text{Step}_\text{cur} < \text{Step}_\text{max}\) then

\[
\text{Step}_\text{cur} \leftarrow \text{Step}_\text{cur} + \text{Step}_\text{unit}
\]

end if

if \(\text{Step}_\text{unit} \leq \text{Ratio}_\text{cur} \leq 1.0 - \text{Step}_\text{unit}\) then

\[
\text{Ratio}_\text{cur} \leftarrow \text{Ratio}_\text{cur} + \text{dir} \times \text{Step}_\text{unit}
\]

end if

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 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.

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

The first and second term estimate the benefit from offloading load instructions as described in Section 3.2. The Benefit 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 OffLD. BEG instruction as described in Section 3.2. The Benefit calculated is substituted into GPUPtrafficReduction 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. 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.

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
