Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems

Kevin Hsieh[‡] Eiman Ebrahimi[†] Gwangsun Kim[*] Niladrish Chatterjee[†] Mike O'Connor[†]
Nandita Vijaykumar[‡] Onur Mutlu[§‡] Stephen W. Keckler[†]
[‡]Carnegie Mellon University [†]NVIDIA [*]KAIST [§]ETH Zürich

ABSTRACT

Main memory bandwidth is a critical bottleneck for modern GPU systems due to limited off-chip pin bandwidth. 3D-stacked memory architectures provide a promising opportunity to significantly alleviate this bottleneck by directly connecting a logic layer to the DRAM layers with high bandwidth connections. Recent work has shown promising potential performance benefits from an architecture that connects multiple such 3D-stacked memories and offloads bandwidth-intensive computations to a GPU in each of the logic layers. An unsolved key challenge in such a system is how to enable computation offloading and data mapping to *multiple* 3D-stacked memories *without burdening the programmer* such that *any application* can transparently benefit from near-data processing capabilities in the logic layer.

Our paper develops two new mechanisms to address this key challenge. First, a compiler-based technique that automatically identifies code to offload to a logic-layer GPU based on a simple cost-benefit analysis. Second, a software/hardware cooperative mechanism that predicts which memory pages will be accessed by offloaded code, and places those pages in the memory stack closest to the offloaded code, to minimize off-chip bandwidth consumption. We call the combination of these two programmer-transparent mechanisms TOM: Transparent Offloading and Mapping.

Our extensive evaluations across a variety of modern memory-intensive GPU workloads show that, without requiring any program modification, TOM significantly improves performance (by 30% on average, and up to 76%) compared to a baseline GPU system that cannot offload computation to 3D-stacked memories.

1. Introduction

Main memory bandwidth is a well-known critical bottleneck for many GPU applications [21, 44, 56]. Emerging 3D-stacked memory technologies offer new opportunities to alleviate this bottleneck by enabling very wide, energy-efficient interfaces to the processor [29, 30]. In addition, a logic layer within a 3D memory stack provides the opportunity to place processing elements close to the data in memory to further improve bandwidth and reduce power consumption [12, 26, 59]. In these near-data processing (NDP) systems, through-silicon vias (TSVs) from the memory dies can provide greater bandwidth to the processing units on the logic layer within the stack, while simultaneously removing the need for energy-consuming and long-distance data movement between chips.

Recent work demonstrates promising performance and energy efficiency benefits from using near-data processing in GPU systems [55, 60]. Figure 1 shows a high level diagram of an example near-data processing system architecture. This system consists of 1) multiple 3D-stacked memories, called memory stacks, each of which has one or more streaming multiprocessors (SMs) on its logic layer, and 2) the main GPU with multiple SMs. Offloading computation to the logic layer

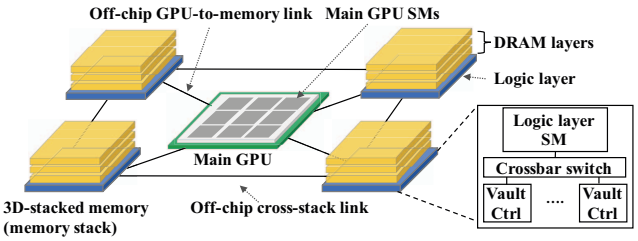

Figure 1: Overview of an NDP GPU system.

SMs reduces data traffic between the memory stacks and the main GPU, alleviating the off-chip memory bandwidth bottleneck and reducing power consumption of the power-hungry off-chip memory bus. Unfortunately, there are two key challenges in such NDP systems that need to be solved to effectively exploit the benefits of near data processing. To solve these challenges, prior works required *significant programmer effort* [2, 55, 60], which we aim to eliminate in this work.

Challenge 1. *Which operations should be executed on the SMs in the main GPU versus the SMs in the memory stack?* Instructions must be steered to the compute units they most efficiently execute on. For example, memory-intensive blocks of instructions could benefit from executing at the logic layer of memory stacks that hold the data they access, while compute-intensive portions could benefit from remaining on the main GPU. Although programmers may have such knowledge, it would be a large burden for them to designate the most appropriate execution engine for all parts of the program, which may change dynamically due to program phase behavior and different input sets.

Challenge 2. *How should data be mapped to different 3D memory stacks?* In a system with multiple memory stacks, such as the one in Figure 1, an application's data is spread across multiple memory stacks to maximize bandwidth utilization of the main GPU. However, the efficiency of an NDP operation primarily depends on whether the data accessed by the offloaded operation is located within the *same* memory stack. We thus need to map data in a way that: 1) maximizes the code/data co-location for NDP operations, and 2) maximizes bandwidth utilization for the code executing on the main GPU. Doing so is challenging because different code blocks and different threads in a program access different parts of data structures at different times during program execution. Determining which part of memory is accessed by which code block instances is difficult, and requiring the programmer to do this places a large burden on the programmer.

Our goal is to solve both challenges *transparently to the programmer*. To this end, we develop two new mechanisms, the combination of which we refer to as TOM (Transpar-

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.27

ent Offloading and Mapping). To solve Challenge 1, we propose an *offload candidate selection mechanism* that can be implemented as a static compiler analysis pass requiring no programmer intervention. The key idea is to statically identify code blocks with maximum potential memory bandwidth savings from offloading to the near-data compute units. The bandwidth savings obtained with compute offloading is a function of the memory intensity of the offloaded code and the bandwidth spent to transfer live-in/live-out registers to/from code being executed on the logic layer. Our mechanism statically estimates memory bandwidth savings of different instruction code blocks to identify the best candidates to offload to NDP compute units in the logic layer of 3D memory stacks (Section 3.1). Our system then dynamically decides whether or not to actually offload the selected code blocks (Section 3.3), based on dynamic system conditions such as SM and bandwidth utilization.

To solve Challenge 2, we propose a *programmer-transparent data mapping* mechanism that places data in the same memory stack as that of the offloaded code that accesses it. This mechanism is based on the key observation that a significant fraction (85% in our experiments) of offloaded code blocks exhibit repeatable memory access patterns. We leverage this repeatability in access patterns to perform simple changes to physical memory mapping to place offloaded code and the data it accesses in the same memory stack. Our mechanism uses the main GPU to evaluate different simple memory mapping options at run time. It then predicts memory pages that the offloaded code block will access, finds the mapping that would keep that data closest to the code that will access it, and maps the predicted pages using the identified best mapping while mapping all *other* pages using the mapping that favors the main GPU.

Contributions. We make the following contributions:

- We propose a new compiler-based mechanism to select instructions to offload to near-data compute units, without requiring any programmer intervention. Our mechanism identifies the best candidate code blocks for offloading by statically estimating the potential memory bandwidth savings for different code blocks. We propose a new runtime mechanism that decides whether or not the selected instructions should be offloaded based on system conditions.
- We propose a new programmer-transparent data mapping mechanism to co-locate offloaded code and data in the same memory stack by exploiting predictability in memory access patterns in offloaded code blocks. Our mechanism retains the memory mapping for all other data to maximize memory bandwidth of code executing on the main GPU.
- We comprehensively evaluate the combination of our two mechanisms, collectively called TOM (Transparent Offloading and Mapping), using 10 memory-intensive general-purpose GPU applications across a variety of system configurations. We show that TOM, on average, improves system performance by 30% (up to 76%), reduces off-chip memory traffic by 38% (up to 99%), and reduces energy consumption by 11% (up to 37%), over a baseline GPU system that cannot offload computation to 3D-stacked memories.

2. Motivation

The first challenge for NDP systems is deciding which code to offload. Most prior research on near-data processing requires the programmer to identify and specify which code will be run close to memory [2, 14, 55] and can ignore the memory intensity of the code [60]. These approaches lead to increased programmer effort, or suboptimal performance by executing some compute-intensive code on the 3D-stack logic layer. In contrast, our approach identifies the candidate code blocks for offloading via static compile-time analysis that maximizes memory bandwidth savings. Figure 2 shows the ideal performance improvement of our proposed offload candidate identification mechanism (described in detail in Section 3.1). With an *idealized system* where there is no overhead for offloading code blocks and where all offloaded code and data are co-located, our static approach has the potential to improve performance by 1.58× on average (up to 2.19×) across a range of 10 memory intensive GPGPU workloads.

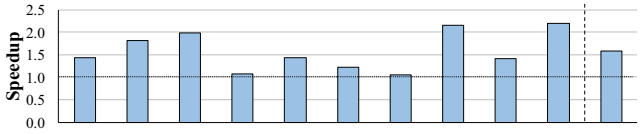

Figure 2: Ideal speedup with near-data processing.

A second challenge is co-locating the offloaded computation with its data, while at the same time not degrading performance of the non-offloaded code due to potentially increased memory stack contention. This is a problem that static compiler analysis inherently cannot solve. The mapping of memory to different stacks is determined by several dynamic components, e.g., the host-side driver, GPU runtime, memory controller, etc. Most prior NDP proposals introduce intrusive program changes such as low level APIs or program annotations to address the data mapping problem [55, 60]. Such approaches make the benefit of NDP available only to those willing and able to re-write their programs, unlike the programmer-transparent approach we propose in this paper. Figure 3 shows how an *ideal mapping*, which simply uses the best two consecutive address bits to map memory pages to memory stacks, can improve the performance of an NDP system.[1] The graph shows that such a simple ideal address mapping of data to memory stacks, which maximizes offloaded code/data co-location, improves performance by 13% on average compared to a state-of-the-art GPU memory mapping policy [9]. These two motivating studies illustrate that there are significant gains that can be achieved by developing intelligent code offloading and data mapping strategies.

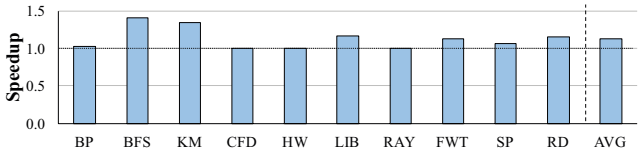

Figure 3: Effect of ideal memory mapping on near-data processing performance.

[1] Section 5 describes our evaluation methodology.

3. Mechanism

We describe our new mechanisms to enable programmer-transparent near-data processing in GPU systems. Our proposal, Transparent Offloading and Mapping (TOM), consists of two key components. The first is a compiler-based technique to identify candidate code blocks for offloading based on cost-benefit analysis of memory bandwidth (Section 3.1). The second component is a software/hardware cooperative mechanism that maps memory pages accessed by offloaded code to where the code will execute, by exploiting common memory access patterns (Section 3.2). After describing these two key components separately, we describe a runtime mechanism that dynamically determines whether an offloading candidate block should really be offloaded (Section 3.3).

3.1. Identification of Offloading Candidates

The objective when identifying candidate instruction blocks for offloading is to improve performance by reducing the main GPU's memory bandwidth consumption. The key to this identification is determining whether offloading a block *saves more memory bandwidth during offloaded execution than it costs in additional data transfers to initiate and complete the offload.* Memory bandwidth savings come from executing offloaded loads/stores in the memory stack. The overhead of offloading is due to transferring the context required by the block to execute in memory, and returning execution results to the main GPU. This section details the process.

3.1.1. Estimating the memory bandwidth cost-benefit. As Figure 1 shows, the main GPU and the memory stacks are connected by unidirectional high speed links similar to the Hybrid Memory Cube (HMC) architecture [34]. Load instructions send addresses through the transmit channels (TX, from the GPU to the memory stack), and receive data back on the receive channels (RX, from the memory stack to the GPU). Store instructions send store addresses and data through the TX channels, and get the acknowledgment messages back from the RX channels. Without loss of generality, we assume the size of address, data, and registers is 4× the size of an acknowledgment message. If loads and stores are executed independently for each thread, we can estimate the change in bandwidth consumption caused by offloading a block as:

$$BW_{TX} = REG_{TX} - (N_{LD} + 2 \cdot N_{ST}) \quad (1)$$

$$BW_{RX} = REG_{RX} - (N_{LD} + 1/4 \cdot N_{ST}) \quad (2)$$

REG_{TX} and REG_{RX} are the number of registers transmitted and received from the memory stacks respectively. These represent the bandwidth cost of offloading the block. The bandwidth benefit of offloading is based on the number of loads, N_{LD}, and stores, N_{ST}, executed in the block.

Equation (1) is derived assuming each load transmits an address, and each store transmits both its address and data through the TX channel. Similarly, Equation (2) is derived by assuming each load gets its data and each store gets its acknowledgment message back on the RX channel.

In a GPU, threads are executed in lock-step warps, so it is straightforward for the hardware to offload code block instances at the granularity of a warp as opposed to the granularity of a single thread. Offloading at warp granularity makes Equations (1) and (2) overly simplistic because, in reality, loads and stores are coalesced by the load-store unit and caches. Furthermore, for loads, the sizes of address and data are different because data is fetched at cache line granularity. To address these issues, we estimate the bandwidth change at warp granularity as follows:

$$BW_{TX} = (REG_{TX} \cdot S_W) - (N_{LD} \cdot Coal_{LD} \cdot Miss_{LD} + N_{ST} \cdot (S_W + Coal_{ST})) \quad (3)$$

$$BW_{RX} = (REG_{RX} \cdot S_W) - (N_{LD} \cdot Coal_{LD} \cdot S_C \cdot Miss_{LD} + 1/4 \cdot N_{ST} \cdot Coal_{ST}) \quad (4)$$

In Equations (3) and (4), S_W is the size of a warp (e.g. 32) and S_C is the ratio of the cache line size to the address size (e.g. 32 for 128B cache lines and 4B addresses). $Coal_{LD}$ and $Coal_{ST}$ are the average coalescing ratios for loads and stores respectively. For example, if all loads in a warp can be coalesced into two cache line accesses on average, $Coal_{LD}$ is 2. Also, $Miss_{LD}$ is the cache miss rate for loads and is accounted for as a co-efficient for the number of loads, N_{LD}, when calculating the bandwidth benefit of offloading.

We propose identifying offload candidate blocks with static compile time analysis since determining instruction dependencies (i.e., REG_{TX} and REG_{RX} values) at run time would introduce high hardware complexity. The compiler can easily determine these terms as they are needed for register allocation and instruction scheduling. However, the compiler does not statically know coalescing ratios ($Coal_{LD}$, $Coal_{ST}$) or cache miss rates ($Miss_{LD}$). We use a conservative estimate for these values so that the identified candidate blocks are most likely beneficial. As such, we assume all memory instructions in a warp are perfectly coalesced so both coalescing ratios are 1. Since GPU cache miss rates are usually high, we choose an estimate of 50% for $Miss_{LD}$, close to the GPU cache miss rates reported by prior works on a wide range of workloads [1, 45].[2]

3.1.2. Offloading candidate block identification. The compiler identifies an instruction block as a potential offloading candidate if the total estimated change in bandwidth as a result of offloading ($BW_{TX} + BW_{RX}$ of Equations (3) and (4)) is negative. This means the benefits of offloading that candidate outweigh the costs and thus offloading is expected to save overall memory bandwidth. The compiler tags each candidate with a 2-bit value indicating whether offloading it is estimated to save RX bandwidth and/or TX bandwidth. Section 4.2 describes how the hardware uses this information to dynamically determine whether or not the candidate should be actually offloaded.

3.1.3. Loops and conditional offloading candidates. In candidate blocks that encapsulate a loop structure, the loop's execution count is a multiplier into the number of loads/stores for the block's bandwidth change calculation. While the overhead of offloading a loop is constant and is proportional to the number of live-in registers required by the block and

[2] While using more aggressive values identifies more offloading candidates, we do not observe clear performance benefits in our experiments. This is because not all aggressively-chosen offloading candidates result in memory bandwidth savings.

the number of registers the block calculates values for, the benefit is determined by the number of executed iterations.

There are three cases for the compiler to handle a loop. First, if the loop trip count can be determined statically, the compiler uses it to determine whether or not it is an offloading candidate. Second, if the loop trip count can be determined before entering the loop at runtime, the compiler marks the corresponding loop as a "conditional offloading candidate". The compiler then provides the condition as a hint to the hardware. Using this hint, hardware decides whether or not to offload the candidate at runtime. For example, the condition of offloading can be a certain register's value (e.g., the loop count) being greater than some threshold. Hardware offloads these conditional offloading candidates only when the condition holds true. Third, if the loop trip count is determined during execution, the compiler conservatively assumes the count to be one and makes the offloading decision based on the loop body. If offloading the loop body is beneficial, the loop is marked as an unconditional offloading candidate.

3.1.4. Offloading candidate block limitations. We impose three limitations on candidate blocks. First, there should not be any on-chip shared memory accesses within a candidate block, as the compute units in memory stacks cannot access the main GPU's shared memory without going through the off-chip link. Second, if the candidate code involves divergent threads, they must converge at the end of offloaded execution. Since the GPU uses a SIMT execution model, threads in a warp may diverge at control flow instructions. Allowing control flow divergence after the execution of an offloaded block can significantly complicate the management of the control divergence/reconvergence stack (e.g., by requiring it to be distributed across the main GPU and the memory stacks). As such, to make sure the threads in a warp converge at the end of an offloaded block's execution, the compiler makes sure the destinations of all control flow instructions are still confined within the candidate block. Third, we do *not* allow memory barrier, synchronization, or atomic instructions in candidate blocks as we do not support synchronization primitives between the main GPU and the logic layer SM. Section 4.4.2 describes the details.

3.1.5. Offloading candidate block examples. Figure 4 shows the example offloading candidate blocks in a sample GPU workload: LIBOR Monte Carlo [6,18]. In this code, there are two loops that are conditional offloading candidate blocks. Each loop has five input values (REG_{TX}, marked as red), one load, and one store (both circled). If the compiler does not take into account loops, it would not select these two loops as offloading candidates with our conservative estimate for cache miss rate and coalescing ratios ($BW_{TX}+BW_{RX} = +110.25$ with 50% cache miss rate and perfect load/store coalescing). However, by considering loops as described in Section 3.1.3, these two loops become conditional offloading candidates. With the same conservative assumptions, this loop would save memory bandwidth if it iterates four or more times as it executes more loads and stores ($BW_{TX} + BW_{RX} = -39$ when it iterates four times).

```
float portfolio_b (float *L, float *L_b) {
    int m, n; float b, s, swapval, v;

    for (n = 0; n < Nmat; n++)
      L_b[n] = -v * delta /(1.0 + delta * L[n]);

    for (n = Nmat; n < N; n++)
      L_b[n] = b * L_b[n];

    return v;
```

Figure 4: Example offloading candidate blocks from LIBOR Monte Carlo [6, 18].

3.2. Programmer-transparent Data Mapping

The goal of programmer-transparent data mapping is to improve code/data co-location in a simple fashion. We first analyze the candidate block's memory access traces to find opportunities towards this goal. Based on our findings, we propose a new automatic mechanism to find an effective data mapping.

3.2.1. Memory access pattern analysis. We observe that a significant fraction of offloading candidates exhibit a very predictable access pattern: *fixed offset*, which means accesses are separated with a constant address offset/distance from each other. Such accesses are predictable and can be used to map memory chunks with that offset onto the same stack. Furthermore, when the offset between accesses has a factor that is a power of two, some least significant bits in the accessed addresses are always the same. Therefore, we can ensure all accesses go to the same memory stack if we use only the least significant N bits to determine the stack mapping. For example, the first loop in Figure 4 accesses two arrays with the same index, and their offset is solely determined by the distance between the arrays' base addresses. As memory is allocated at a page granularity, which is a power of two value, this distance usually has a factor that is a power of two.

To understand how often such an access pattern happens, we categorize offloading candidates (chosen by the mechanisms described in Section 3.1) based on the percentage of fixed-offset memory accesses. Figure 5 shows the results. We make two observations based on the results.

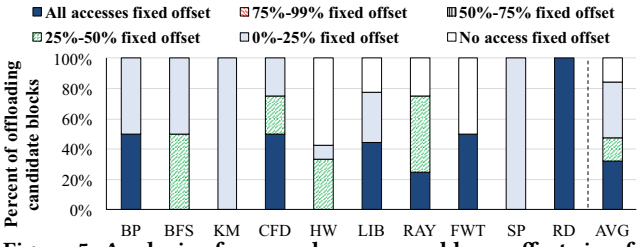

Figure 5: Analysis of accessed memory address offsets in offloading candidates.

First, 85% of all offloading candidates in the GPU workloads we studied have some memory accesses with fixed offset that can be used to improve data locality. Second, six workloads have offloading candidates that *always* access memory with a fixed offset. This means we can find a simple mapping for these workloads that would keep a large portion of accesses within the same stack as offloaded blocks.

We take advantage of these observations by using consecutive bits in the memory address to determine memory stack

address mapping. We avoid choosing bits from the *cache line offset* to ensure off-chip link efficiency and main memory row buffer locality are not reduced. Assuming fixed offset addresses have a common factor of 2^M and the cache line size is 2^N, our best bit position choices for mapping to different memory stacks are among bits $[M-1:N]$.

To demonstrate the effectiveness of using consecutive address bits for mapping addresses to memory stacks, we sweep all consecutive 2-bit mappings[3] in a system with 4 memory stacks and compare the compute/data co-location of the best 2-bit mapping with the baseline GPU mapping [9]. As we offload each candidate block at the granularity of a warp, an *offloading candidate instance* is a warp that executes an offloading candidate block. We define the compute/data co-location as the *probability of accessing one memory stack in an offloading candidate instance*. Higher probability indicates higher compute/data co-location. Our evaluation shows that using the best consecutive bit position choice, the probability of accessing one memory stack from an offloading candidate instance goes up from 38% (with baseline data mapping) to 75% on average (almost a 2× improvement). This shows the effectiveness of using a simple memory mapping mechanism to increase the compute/data co-location to maximize the benefits of NDP.

3.2.2. Predictability of the best memory mapping. Adjusting memory mapping after data has been placed in GPU memory and during kernel execution can involve high overhead in moving data between stacks and could easily eliminate the benefits of NDP. To address this, we propose a mechanism to *predict the best memory mapping* before data is placed in memory.

We find that we can predict the best memory mapping by observing a small number of initial offloading candidate instances' memory behavior. Figure 6 shows how close we can get by making a choice on the memory mapping to use by observing only the first 0.1%, 0.5%, and 1% offloading candidate instances. The figure shows that the mapping chosen after observing only 0.1% of offloading candidate instances achieves a probability of accessing one memory stack of 72%, which is only 3% less than that obtained with oracle knowledge. This is intuitive because most GPU programs access memory based on the index of threads and thread blocks, which makes access patterns very predictable and consistent among threads [38]. We conclude that we can predict a simple memory stack physical address mapping that significantly improves code/data co-location by observing a small number of initial offloading candidate instances.

3.2.3. Programmer-transparent data mapping. Based on Section 3.2.2, we propose a software/hardware cooperative mechanism to improve compute/data locality automatically, which we call **programmer-transparent data mapping**. The key idea is to learn the best memory mapping by observing a small number of initial offloading candidate instances, and apply this mapping *only* to the data that offloading candidate blocks access.

[3]We only need 2 bits to determine the memory stack in a system with 4 memory stacks. The result of the sweep starts from bit position 7 (128B GPU cache line size) to bit position 16 (64 KB). Based on our results, sweeping into higher bits does not make a noticeable difference.

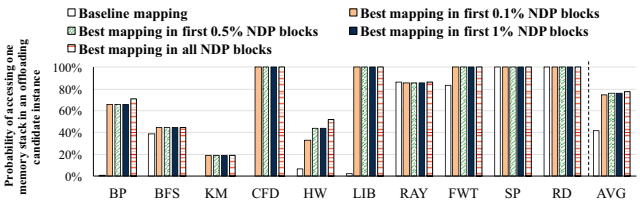

Figure 6: Effectiveness of the best memory mapping chosen from different fraction of offloading candidate instances.

With our mechanism, GPU kernel execution is split into a relatively short *initial learning phase* followed by regular execution. During the learning phase, all GPU kernels run on the main SMs and the corresponding data resides in the CPU memory while the mechanism learns the best memory mapping for data accessed by offloading candidates. After the learning phase is over, before regular execution commences, we copy data accessed by offloading candidates from the CPU memory to the GPU memory based on the learned best memory mapping. Our mechanism essentially *delays* the memory copy from the CPU to the GPU in the typical GPU execution flow so there is no extra data remapping overhead. During regular execution, offloaded code runs on memory stack SMs while the data it accesses is mapped using the best mapping discovered in the learning phase. As we apply the best mapping only to the data accessed by the offloading candidates, all the other data is still mapped with the sophisticated memory mapping scheme typically used by GPUs [9], which maximizes memory bandwidth for the code the main GPU executes. Section 4.3 describes the detailed flow of programmer-transparent data mapping.

3.3. Dynamic Offloading Aggressiveness Control

Aggressively offloading candidates determined by the compiler to memory stacks may make the system slower in two scenarios. First, memory stack SMs could become new performance bottlenecks if the number of offloading requests is more than what they can handle. As each SM has a limit on concurrent warps running on it, the SMs in memory stacks cannot spawn new warps for an offloaded block after the number of concurrent warps reaches the hardware limit. When this happens, SMs in the main GPU would be waiting for the offloaded blocks to complete, but these offloaded blocks would not be making progress as they are waiting to be spawned. Second, there could be a discrepancy in the bandwidth savings of the RX and TX off-chip links. Offloading a block may save significant bandwidth in the RX channel, but introduce additional traffic in the TX channel. If the TX channel is the performance bottleneck for an application, offloading such a block will exacerbate the problem.

To address the above cases, we propose **dynamic offloading aggressiveness control**, which uses run-time information to make the final decision on whether or not each candidate should be offloaded. Our dynamic offloading control mechanism may override the compiler's offloading recommendation in two ways. First, the GPU keeps track of the number of pending offloading requests sent to each memory stack, and stops further offloading when that number

reaches the number of warps that can concurrently execute on the corresponding memory stack SM. Doing so prevents the over-offloading situation described above. Second, the GPU monitors the bandwidth utilization of both TX and RX channels, and does not offload blocks that would introduce more traffic to a channel that is already above some threshold utilization rate. As described in Section 3.1, the compiler tags each offloading candidate block with a 2-bit value to indicate its bandwidth savings in the TX and RX channels. The hardware uses this information to make offloading decisions at run time.

4. Implementation

In this section, we describe the implementation details of our proposal, TOM. We first introduce the hardware components added by our proposal (Section 4.1). We then provide detailed design of the NDP offloading (Section 4.2) and the programmer-transparent data mapping (Section 4.3) mechanisms. Finally, we discuss how we handle two important design considerations, virtual address translation (Section 4.4.1) and cache coherence (Section 4.4.2).

4.1. Hardware Block Diagram

Figure 7 presents the high level block diagram that shows how our proposals for NDP offloading fit within a typical GPU pipeline [20, 37]. We add three new components to support our mechanisms: (1) an Offload Controller to make final offloading decisions, (2) a Channel Busy Monitor to monitor the utilization of the off-chip channels, and (3) a Memory Map Analyzer to support programmer-transparent data mapping, as described in Section 4.3. We briefly describe each component and explain how it fits into the NDP offloading event flow.

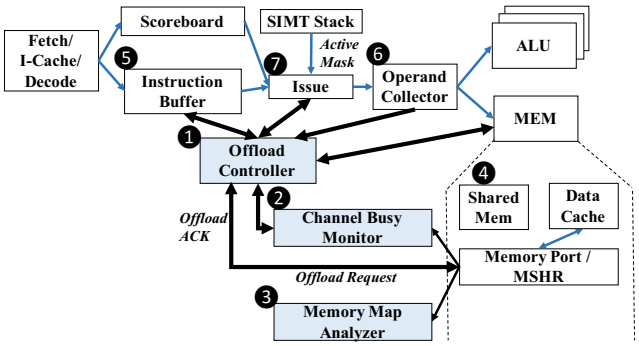

Figure 7: Block diagram of the NDP hardware and its interactions with the GPU pipeline.

Offload Controller (❶). The Offload Controller works with the main GPU pipeline to provide three functions. First, it determines whether a candidate block should be offloaded based on runtime information. This runtime decision involves handling of conditional offloading candidates (Section 3.1.3) and dynamic offloading aggressiveness control (Section 3.3). Second, it packs the offloading information and sends it to the memory stack SMs. Third, it resumes the offloaded warp when it receives the corresponding acknowledgment packet from memory stack SMs.

Channel Busy Monitor (❷). The Channel Busy Monitor tracks the utilization of off-chip TX/RX channels. When the utilization rate of a channel reaches a pre-defined threshold, it reports the channel as busy to the Offload Controller (❶).

Memory Map Analyzer (❸). The Memory Map Analyzer is set up by the GPU Runtime to provide the number of memory stacks accessed by each offloading candidate instance for all different potential stack mappings. This unit is enabled only during the learning phase of programmer-transparent data mapping, as we describe further in Section 4.3.

4.2. Design of NDP Offloading

Interface between the compiler and the hardware. The compiler provides information about offloading candidates to the hardware in two ways. First, we introduce a new instruction in the ISA that is used by the compiler to indicate the beginning of an offloading candidate block to the hardware. Second, the compiler provides an **offloading metadata table** in the program to the hardware. Each entry in this table is associated with an offloading candidate, and provides the begin/end PC addresses, live-in/live-out registers, 2-bit tags to indicate the TX/RX channel savings (Section 3.1.2), and the condition for conditional offloading candidates (Section 3.1.3). This table is allocated by the compiler and placed in on-chip Shared Memory (❹).

Offloading candidate block detection. In the pipeline, each instruction is decoded and placed into the Instruction Buffer (❺). When the Instruction Buffer detects the instruction as the beginning of an offloading candidate block, it marks this warp as not ready, and consults the Offload Controller (❶) for an offloading decision. The Offload Controller fetches its offloading metadata from Shared Memory (❹), and uses it to make an offloading decision.

Dynamic offloading decision. The dynamic offloading decision consists of three steps: (1) The Offload Controller checks whether the condition for offloading is true for a conditional offloading candidate. It does so by getting the corresponding register value from the Operand Collector (❻) and performing the value comparison based on the corresponding condition (Section 3.1.3). (2) If one of the TX/RX channels is signaled as busy by the Channel Busy Monitor (❷) and the 2-bit tag for this block indicates that it would introduce more memory traffic to a busy channel, the Offload Controller does *not* offload it. (3) The Offload Controller determines the offloading destination based on the memory stack that will be accessed by the first instruction of the block.[4] It then checks whether the number of pending offloading requests to that memory stack have reached the warp limit of the memory stack SM, and, if so, skips offloading (Section 3.3). Note that the pipeline can schedule and execute other warps while waiting for the offloading decision.

Sending offloading requests. After the decision is made, the warp that is waiting for the offloading decision becomes

[4]If the first instruction of the offloading candidate is not a memory instruction, the Offload Controller saves the live-registers and asks the pipeline to execute until the first memory instruction. When the first memory instruction is encountered and the destination stack is determined, offloading *actually* happens and the offloaded block restarts execution from the beginning of the block on the appropriate memory stack SM. The redundant execution at the main GPU does not affect the program state because we update the live-out registers only after offloading completes.

ready. The Issue unit (❼) issues the instructions into the pipeline as usual if the decision is *not to* offload the block. Otherwise, it sends the instruction to the Offload Controller. The Offload Controller packs live-in registers, begin/end PCs, and active masks as an offloading request and sends it to the memory stack. In our evaluation, we assume the pipeline latency of offloading a task is 10 cycles. This latency is small compared to the at least 200 cycle latency of each memory access within an offloaded block, which contains many such memory accesses. The GPU pipeline can execute other warps while waiting for the offloaded tasks.

Receiving offload acknowledgment. When an offloaded block completes its execution on the memory stack, the memory stack SM sends an offload acknowledgment packet to the main GPU, which includes live-out registers and the cache lines that need to be invalidated (Section 4.4.2). The Offload Controller (❶) requests cache invalidations and register updates, and then restarts the corresponding warp from the next instruction of the end of the block on the GPU.

4.3. Design of Programmer-transparent Data Mapping

Our programmer-transparent data mapping is a software/hardware cooperative mechanism to choose and apply the best memory mapping for compute/data co-location based on the access pattern of a small number of initial offloading candidate instances (Section 3.2.1). In hardware, we add a **memory mapping analyzer** in the GPU (❸ in Figure 7). In software, we modify the **GPU host-side driver** that runs on the CPU and the **GPU runtime** that runs on the GPU. Existing applications can benefit from our programmer-transparent data mapping without any program modification.

This mechanism consists of the following steps:

1. Before launching a GPU kernel, a GPU application needs to allocate memory through the GPU driver on the CPU, and copy corresponding locations from the CPU memory into the GPU memory. With programmer-transparent data mapping, the GPU driver still allocates memory in the GPU virtual memory space[5], but *delays the copy by initially mapping the GPU virtual memory to CPU memory* During the initial learning phase, the GPU driver records each memory allocation in a **memory allocation table**[6] for further reference.

2. When the application requests a GPU kernel launch, the GPU driver uses the GPU runtime to set up the memory mapping analyzer (❸ in Figure 7) based on two tables: the **memory allocation table** from Step 1 and the **offloading metadata table** from the compiler (Section 4.2). The GPU driver then launches the kernel as usual. Since the GPU virtual memory has been allocated in CPU memory, during the relatively short initial learning phase, the GPU accesses the memory through the GPU-to-CPU link (i.e., PCI-E). As we describe in Section 5, our experimental setup faithfully accounts for this extra latency during the learning phase.

3. The memory mapping analyzer monitors the execution of GPU threads and their memory accesses. By doing so, it calculates how many memory stacks would be accessed by each offloading candidate instance for all different potential stack mappings (e.g., using bits 7:8, 8:9, ..., 16:17 in a system with four memory stacks). Separately, for each application-allocated memory range (Step 1) that has been accessed by an offloading candidate, the analyzer sets a bit in the memory allocation table to indicate that this range should apply the memory mapping that favors offloaded blocks.

4. When the memory mapping analyzer has seen the predetermined number of offloading candidate instances (e.g., 0.1% of all instances) , it issues an interrupt to the GPU runtime. The GPU runtime stops execution on all SMs in the GPU and uses the data recorded by the mapping analyzer to determine the best memory mapping. This is the mapping that leads to the most accesses to the stack that the offloaded block would execute on.

5. Finally, the GPU runtime requests the GPU driver on the CPU to perform the memory copy. In the baseline, this would have happened *before* the kernel launch. In our proposal, however, the GPU driver copies the memory ranges that have been accessed by an offloading candidate block into the GPU memory, using the best found mapping in the learning phase. Other memory ranges are copied over using the default mapping that favors the code running on the main GPU. Subsequently, regular execution resumes on the GPU.

4.4. Design Considerations

There are two considerations that are important to the efficiency and correctness of NDP offloading: 1) virtual address translation in memory stacks, and 2) cache coherence between the SMs in the main GPU and the SMs in the memory stacks. We address these here.

4.4.1. Virtual address translation. The GPU application works with virtual addresses that are translated to physical addresses using a hardware TLB and an MMU that accesses page tables in memory. We assume memory stack SMs are equipped with similar TLBs and MMUs and are capable of performing virtual address translation. According to our evaluation, the size of the MMU and TLB per SM is fairly small: 1-2K flip-flops and small amount of logic. This accounts for less than 2% of the area of a memory stack SM.

Providing this capability to the memory stack SMs can pose two challenges: First, the page table needed for address translation may not be located in the same memory stack as the requesting SM. Such accesses would utilize cross-stack links used for remote data access already present in our architecture (as shown in Figure 1 and described further in Section 5). Second, if the GPU SMs were to update the page table, a TLB shootdown may be needed to maintain the correctness of address translation. However, we offload candidate blocks only *after* we complete the memory copy and the page table update. Since we set up all page tables before offloading, there is no need for memory stack TLB shootdowns.

4.4.2. Cache coherence. Both the SMs in the GPU and the SMs in the memory stacks have caches to exploit locality and coalesce memory accesses. As blocks get offloaded to memory stack SMs, and data is cached on the SMs in addition to the main GPU caches, correctness issues may occur if stale data is accessed incorrectly. For example, an SM in a memory

[5]Section 4.4.1 provides the details of the virtual address translation.
[6]Section 6.6 provides the details of the memory allocation table.

stack may execute an offloaded block and update some values in memory. Subsequently, if the GPU SMs read stale data from their caches, they will not obtain the most up-to-date data. Therefore, we make sure GPU SMs and memory stacks SMs always observe the most up-to-date values.

A naive solution would be to provide cache coherence between the main GPU's caches and memory stack SM caches. However, extending the traditional cache coherence protocols to the number of caches involved is difficult, potentially requiring large state and interconnect bandwidth [22,24]. The corresponding coherence traffic would need to go through off-chip links between the GPU and the memory stacks, and can consume more bandwidth than what is reduced by offloading.

In our NDP architecture, when the main GPU SM requests offloading of a block, the memory stacks essentially execute some instructions on behalf of the requesting main GPU SM. Since these are the only SMs that execute instructions from the same Cooperative Thread Array (CTA), without loss of generality, we focus on their cache coherence here. In the GPU programming model, there is no ordering or consistency guarantee between threads in different CTAs. Each CTA is bound to a specific SM, and there is no cache coherence requirement across SMs. Additionally, programmers are required to explicitly use memory barriers or thread synchronization instructions to ensure memory access ordering across threads within a CTA [38].

We guarantee the cache correctness of offloaded execution in three steps. First, the requesting SM in the GPU pushes all memory update traffic from itself to memory before issuing the offloading request. This is not difficult because most GPUs employ write through caches [53]. Second, the memory stack SM invalidates its private cache before spawning a new warp for the offloaded block. These two steps guarantee that the memory stack SM observes the most up-to-date data from the requester SM. Third, the memory stack SM records the cache line addresses that have been updated by this offloaded block, and sends these addresses back to the requester SM when the offloaded block exits. The requester SM then invalidates these cache lines in the GPU and subsequently obtains the latest data from memory, if necessary[7]. According to our evaluation, the average performance overhead of guaranteeing correct execution with this mechanism is only 1.2%.

5. Methodology

5.1. System Modeling and Configuration

We model our proposed NDP GPU architecture by modifying GPGPU-Sim 3.2.0 [6] to include the modeling of memory stacks and logic layer SMs. The off-chip links between GPU and memory stacks are modeled as unidirectional high-speed links using BookSim [11], a cycle-accurate interconnection network simulator also used for on-chip interconnect simulation in GPGPU-Sim. We faithfully model all overheads incurred by our proposed NDP architecture, including the traffic from transferring live-in/live-out registers for offloaded blocks and any cache coherence overhead. Since we assume the learning phase of the programmer-transparent data mapping is executed out of the CPU memory (before data is copied over to the GPU memory), all memory accesses in the learning phase are modeled with the measured PCI-E bus latency from Miller et al. [36].

In our architecture, we assume cross-stack links to allow remote data access for the SMs in memory stacks. The bandwidth of each link is $0.5\times$ of each link between the GPU and each memory stack. We assume the internal memory bandwidth in memory stacks is $2\times$ of the link bandwidth between GPU and memory stacks. Compared to some prior 3D-stacked proposals [2,14,60], this is a conservative assumption. However, valid arguments can be made to consider even more conservative design points for the internal memory bandwidth of the stacks. As such, we also evaluate a system configuration that matches internal and external memory bandwidth.[8] That is, the baseline GPU would be able to fully saturate a memory stack's bandwidth in this design point.

Table 1 summarizes our simulation parameters. The GPU configuration and the GPU-to-memory bandwidth are similar to the configuration of NVIDIA Maxwell, a state-of-art GPU architecture. To make a fair comparison between the baseline GPU system and our NDP GPU system, the total number of SMs in these two configurations are the same.

For our energy evaluations, we use GPUWattch [33] to model the power consumption of the SMs (both GPU SMs and memory stack SMs) and on-chip interconnect. We assume the off-chip link consumes 2 pJ/bit for packet transfer, and 1.5 pJ/bit/cycle when it is idle [27]. The power of stacked memory is calculated using the Rambus power model [57], which models the power consumption of TSVs [29] and 3D-stacked DRAM devices [8]. From this model, row activation energy is estimated to be 11.8 nJ for a 4KB row and the DRAM row-buffer read energy is 4 pJ/b.

Our baseline memory mapping is similar to the one used by Chatterjee et al [9]. We spread consecutive cache lines to different memory stacks and vaults to maximize memory level parallelism and load balance across them. Similar to prior work [61], we XOR a subset of higher-order bits to form the memory stack index to prevent pathological access conflicts on a specific memory stack.

5.2. Offloading Candidate Selection Tool

We implement a tool to analyze the PTX (Parallel Thread Execution ISA) files generated by the CUDA compiler [40]. This tool performs the static analysis proposed in Section 3.1. The tool analyzes all potential instruction blocks at the PTX level, and calculates memory bandwidth savings using the equations in Section 3.1. The number of live-in registers is calculated by determining the number of operands that are not

[7] Note that this mechanism may potentially change the observed ordering for other threads in the same CTA. Without offloading, the other threads in the same CTA can observe the data change immediately as they run on the same SM. With offloading, the other threads cannot observe it until the offloaded block completes. However, this is not a problem because cross-thread ordering is not guaranteed without explicit memory barrier/thread synchronization instructions. Since we do not include such synchronization instructions in offloaded candidate blocks, all threads still observe the correct values at the memory barrier or thread synchronization point.

[8] Section 6.5 provides the results of sensitivity studies on the cross-stack link bandwidth and the internal memory bandwidth.

Table 1: Major simulation parameters.

Main GPU	
Core Number	68 SMs for baseline 64 SMs for the NDP system
Core Clustering	4 SMs per cluster
Core Configuration	1.4 GHz, 48 warps/SM 32 threads/warp, 32768 registers 8 CTAs/SM, 48KB shared memory
Private L1 Cache	32KB, 4-way, write through
Shared L2 Cache	1MB, 16-way, write through
Clock Frequency	Interconnect 1.25 GHz, L2 700 MHz
Off-chip Links	
GPU to Memory	80 GB/s per link, 320 GB/s total
Memory to Memory	40 GB/s per link, fully connected
Memory Stack	
SM in Memory Stack	1 SM per memory stack, 48 warps/SM
Memory Stack Configuration	4 memory stacks, 16 vaults/stack 16 banks/vault, 64 TSVs/vault 1.25 Gb/s TSV signaling rate
Internal Memory Bandwidth	160 GB/s per memory stack 640 GB/s total
DRAM Scheduling Policy	FR-FCFS [47, 63]
DRAM Timing	DDR3-1600 [35]

generated using the instruction block itself, and the number of live-out registers is calculated by determining the number of operands that are needed by subsequent instructions. The tool identifies loops by first detecting backward conditional branch instructions. When the condition of the branch is a comparison to a value that is generated within the block with a simple add/subtract operation, the condition is identified as a loop condition. The output of the tool is the list of offloading candidate blocks with their associated begin/end PCs, live-in/live-out registers, 2-bit tags to indicate whether offloading the block would save bandwidth on TX/RX channels, and the condition to offload if it is a conditional offloading candidate. The output of the tool is then fed into our GPGPU-Sim-based performance model. Our tool works on PTX because the evaluation infrastructure, GPGPU-Sim, works only on PTX. The same idea can be directly applied to SASS or any other instruction set that runs on real GPUs.

5.3. Evaluated Applications and Metrics

We evaluate the memory intensive workloads (with memory bandwidth utilization >50%) from Rodinia 3.0 [10] , scientific workloads used by GPGPU-Sim [6], and CUDA SDK [39]. We do not consider compute-intensive workloads, as TOM does not find offloading candidates in these workloads; hence, their performance is not affected. We use the slightly modified K-means from Rogers et al. [48], which replaces texture/constant memory with global memory to use a larger input. Table 2 summarizes our workloads. We run all applications to completion or for 2 billion instructions, whichever comes first. We record the performance metrics at the same point of execution across all configurations.

The major performance metric we use is IPC (Instruction Per Cycle), and we normalize it to a baseline GPU system without NDP. We also present the total memory traffic on all off-chip links (including GPU-to-memory and memory-to-memory) to quantify off-chip bandwidth.

Table 2: Summary of applications.

Name	Abbr.	Name	Abbr.
Back Propagation [10]	BP	BFS Graph Traversal [10]	BFS
K-means [10, 48]	KM	CFD Solver [10]	CFD
Heartwall [10]	HW	LIBOR Monte Carlo [6]	LIB
RAY Tracing [6]	RAY	Fast Walsh Transform [39]	FWT
Scalar Product [39]	SP	Parallel Reduction [39]	RD

6. Results

We evaluate the effect of our proposal, TOM, on application performance, memory traffic, and energy consumption. We evaluate two NDP policies: (i) *NDP-Uncontrolled* or *no-ctrl*: always offloading the candidate blocks and (ii) *NDP-Controlled* or *ctrl*: enabling dynamic offloading control described in Section 3.3. We evaluate these NDP policies along with two memory mapping schemes: (i) *bmap*: baseline GPU memory mapping [9], and (ii) *tmap*: our proposed programmer-transparent data mapping. Unless specified otherwise, the results are over a baseline GPU without NDP and with the baseline memory mapping described in Section 5.

6.1. Effect on Performance

Figure 8 shows the speedup of various NDP policies using different memory mapping schemes, normalized to the baseline system. Three major observations are in order:

1. TOM significantly improves performance. When all TOM techniques are enabled (*NDP-Controlled and tmap*), average performance improves by 30% (up to 76%) over the baseline. We observe speedup for all workloads. As the baseline has the same number of SMs as our NDP configuration, we conclude that automatically moving some computation to memory stacks with TOM provides better performance than having additional SMs at the main GPU.

2. Programmer-transparent data mapping improves performance over baseline memory mapping. On average, NDP with programmer-transparent data mapping (*tmap*) provides an additional 10% speedup over NDP with baseline memory mapping (*bmap*). For some workloads, the improvement from programmer-transparent data mapping is significant. KM's performance improves from 3% to 39% and RD's performance improves from 51% to 76% as a result of better memory mapping. Some workloads are not sensitive to programmer-transparent data mapping because the mapping chosen by our data mapping is very similar to the baseline mapping. There is one workload (BFS) that gets slower with programmer-transparent data mapping (NDP's improvement reduces from 29% to 21%). The reason is that BFS is a very irregular workload, and the mapping chosen by our mechanism with the initial 0.1% offloading candidate instances is not the best one for all offloading candidate instances. Assuming we had oracle knowledge and we chose the optimal mapping, the performance improvement of BFS would be 64% (not shown).[9]

3. Offloading aggressiveness control is an enabler for NDP's performance improvements. Without offloading control (*NDP-Uncontrolled*), the system becomes slower by 3%/7% on average (with *tmap*/*bmap*). This is because the main SMs

[9]This number is different from (and actually higher than) that in Figure 3 for BFS, because the experiments in Figure 3 do *not* include all the mechanisms we propose, whereas the experiments in Section 6 do.

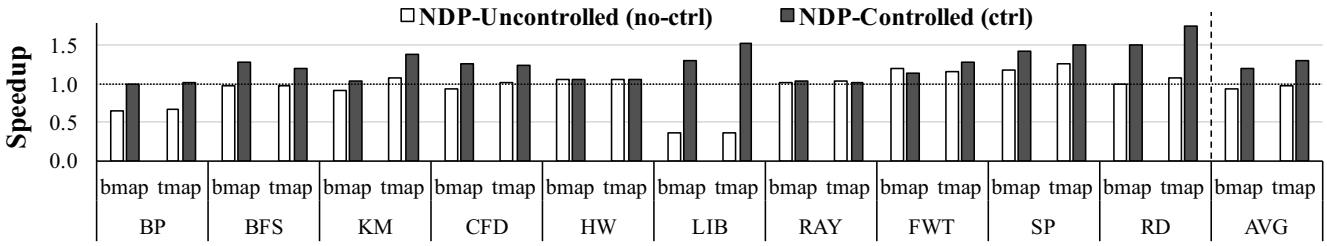

Figure 8: Speedup with different NDP offloading and memory mapping policies.

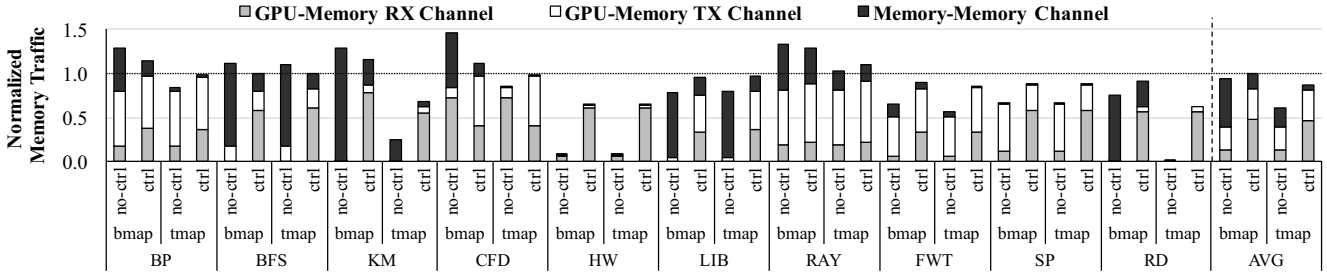

Figure 9: Memory traffic with different NDP offloading and memory mapping policies.

offload more warps than the memory stack SMs can handle. As such, the computational capacity of memory stack SMs becomes the new bottleneck. However using offloading aggressiveness control (*NDP-Controlled*) improves performance by 30% over the baseline, and the workloads that are otherwise slowed down by uncontrolled offloading become significantly faster. For instance, offloading control enables LIB to experience a performance improvement of +52% instead of a degradation of -64%. On average, offloaded instructions reduce from 46.4% (*NDP-Uncontrolled*) to 15.7% (*NDP-Controlled*) of all instructions executed (not graphed). We conclude that the mechanism we propose in Section 3.3 for controlling how aggressively we offload blocks on an NDP architecture is essential to enabling NDP's performance improvements.

6.2. Effect on Memory Traffic

Figure 9 shows the effect of NDP offloading and memory mapping on off-chip memory traffic. The memory traffic is normalized to the baseline, and segmented by the channel category. We make two major observations.

First, as expected, TOM is effective in reducing the overall memory traffic. When the system offloads all candidate blocks with programmer-transparent data mapping (*no-ctrl* and tmap), the total memory traffic reduces by 38% on average (up to 99%). This shows the effectiveness of our automatic offloading candidate selection and programmer-transparent data mapping. With controlled offloading (*ctrl* and *tmap*), the memory traffic reduction is 13% on average (up to 37%). This is much less than the uncontrolled offloading because some memory-intensive offloading candidates are still executed in the GPU. It also implies that if we increase the computational capacity of SMs in memory stacks, we can potentially offload more work to them and save more memory traffic. We discuss this further in Section 6.4.

Second, programmer-transparent data mapping is very effective at reducing memory-to-memory traffic. On average, memory-to-memory traffic is reduced from 55% (bmap) to 22% (tmap) of the total memory traffic in the baseline when we offload all candidate blocks (*no-ctrl*), which is a 2.5× reduction. We observe a similar reduction ratio with controlled offloading (*ctrl* with *bmap* and *tmap*). The memory-to-memory traffic is a smaller proportion in this case because some offloading candidates are executed in the main GPU.

6.3. Effect on Energy Consumption

Figure 10 shows the effect of our NDP architecture on energy consumption. Energy consumption is segmented by the source. We make two major observations.

First, TOM is effective in reducing system energy consumption. Total energy consumption reduces by 11% on average (up to 37%) when all TOM techniques are enabled (*ctrl* + *tmap*). Compared to the baseline, average energy consumption due to SMs reduces from 77% to 68%, while the energy consumption due to off-chip links reduces from 7% to 5%. These energy savings are mainly a result of the performance improvement which reduces leakage energy, and reduced memory traffic over off-chip links.

Second, both programmer-transparent data mapping and offloading aggressiveness control are important to the energy savings. Without them, total energy consumption increases by 8% on average. This is mainly because the system runs slower without these techniques. Even though memory traffic reduces without these techniques, the extra leakage energy due to longer execution time overshadows the savings from transmitting fewer memory packets.

6.4. Sensitivity to Computational Capacity of SMs in Memory Stacks

The performance and traffic results indicate a trade-off between performance improvement and traffic reduction. With offloading aggressiveness control, we see an average performance improvement of 30% with a memory traffic reduction of 13%. However, offloading *all* candidate blocks degrades performance by 3%, but saves 38% memory traffic. With offloading aggressiveness control enabled, TOM does not save

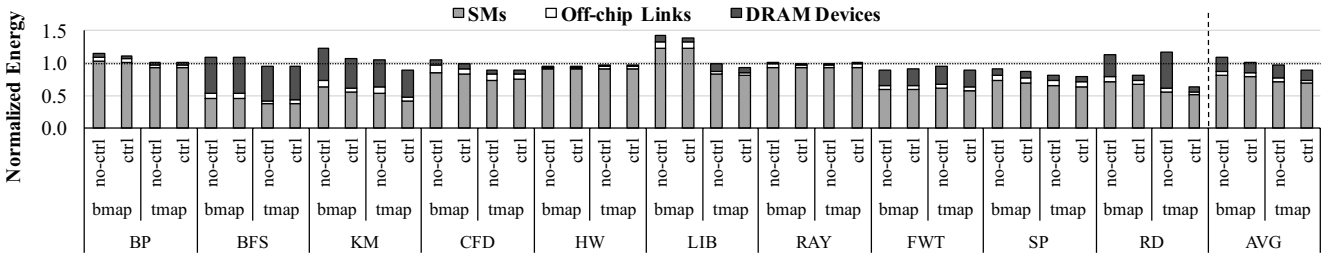

Figure 10: Energy consumption with different NDP offloading and memory mapping policies.

as much memory traffic because some memory-intensive candidate blocks are still executed on the GPU.

We can get the best of both worlds by adding more computational resources (i.e., *warp capacity*) to memory stack SMs because limited warp capacity is the major reason we need offloading aggressiveness control. To increase warp capacity, we need to add more storage for the register file, warp state, and the warp scheduler. The major cost comes from additional registers. To begin with, memory stack SMs require fewer registers per thread than the main GPU SMs since an offloaded block executes only a portion of the code in a kernel. We find that the largest number of live registers in an offloaded block is 26 versus 49 for an entire thread, among all our workloads. This means that if we increased the size of the register file on a memory stack SM to be the same as that on the main GPU, we could potentially run 2× the warps on a memory stack SM. Figures 11 and 12 show the effect of adding warp capacity on performance and memory traffic.

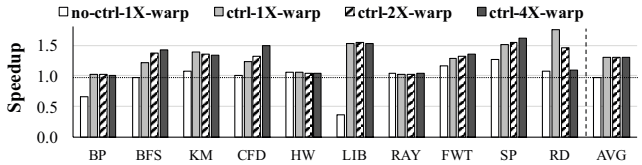

Figure 11: Speedup with different warp capacities in memory stack SMs.

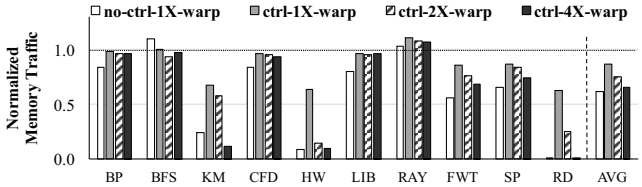

Figure 12: Memory traffic with different warp capacities in memory stack SMs.

As the figures show, adding warp capacity to memory stack SMs saves more memory traffic while maintaining high performance improvements. On average, 4× warp capacity (*ctrl-4×-warp*) saves an additional 20% memory traffic over 1× warp capacity (*ctrl-1×-warp*) while maintaining similar average speedup. The memory traffic savings with 4× warp capacity is 34% compared to the baseline, which is close to the 38% traffic reduction from uncontrolled offloading. Performance also improves by 29% with 4× warp capacity compared to the baseline, and most workloads have better performance than with 1× warp capacity. The only exception is RD, which runs significantly slower with more warp capacity. This is because more than half of the instructions in the offloaded blocks of RD are ALU instructions, so running 4× warps in parallel on the memory stack SMs makes the compute pipelines of these SMs new bottlenecks. A more sophisticated offloading aggressiveness mechanism that considers the ALU instruction ratio would help alleviate this problem, and we leave it as future work. We conclude that increasing warp capacity can maintain high performance improvements while significantly reducing memory traffic.

6.5. Sensitivity to Internal and Cross-Stack Memory Bandwidth

Many NDP proposals based on 3D-stacked memory assume the memory stack compute units can exploit much more internal memory bandwidth from TSVs, and that bandwidth cannot necessarily be fully utilized by the main processor in the system because of cost and speed of off-chip links and pins [2, 14, 60]. In most of our evaluations, we also assumed that the total internal memory bandwidth is 2× the external link bandwidth that links the GPU to each memory stack. However, such an assumption may not always be true, as the benefit of building a memory stack with such high internal bandwidth may not always justify its cost. Thus, we also evaluate our NDP architecture with a configuration where the internal memory bandwidth is equal to the external link bandwidth. Figure 13 presents the performance results.

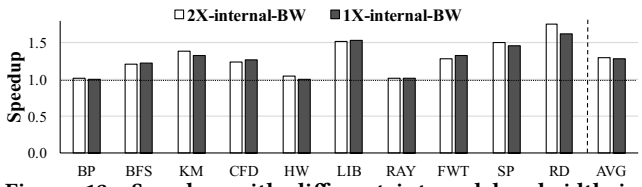

Figure 13: Speedup with different internal bandwidth in memory stacks.

The speedup with equal internal/external memory bandwidth (*1×-internal-BW*) is on average within 2% of that achieved with 2X internal memory bandwidth (*2×-internal-BW*). On average, NDP with equal internal/external bandwidth has 28% speedup over the baseline. As such, the performance of our NDP architecture does not hinge on higher internal bandwidth from TSVs. The reason is that the main GPU is usually bottlenecked by one of the TX/RX off-chip channels. Our NDP offloading enables memory stack SMs to directly access the memory controller with on-package interconnect. These SMs can make use of the internal memory bandwidth when the GPU is bottlenecked by one of the two channels. This makes our NDP architecture more efficient even without extra internal bandwidth in the memory stacks.

Another parameter in our NDP architecture is the memory bandwidth of cross-stack links that allow remote data access

for the SMs in memory stacks. We choose the cross-stack link bandwidth as 0.5× of the GPU-to-stack links by default because the cross-stack bandwidth requirements are smaller. We sweep this ratio of cross-stack link bandwidth to GPU-to-stack link bandwidth. Average performance benefit is 17% for 0.125×, 29% for 0.25× and 31% for 1×. These results show that performance gains of our mechanism are significant for a wide range of design points.

6.6. Area Estimation

The major area overhead of our proposed NDP architecture comes from storage for three components: the memory mapping analyzer (Section 4.1), the memory allocation table, and the offloading metadata table (Section 4.3).

For each offloading candidate instance, the Memory Map Analyzer needs 40 bits to support 10 potential memory mappings in a 4 memory stack system. With 48 warps/SM, we need 40 × 48 = 1,920 bits storage in the memory map analyzer per SM to support all concurrent warps. In the memory allocation table, each entry contains the start address, the length, and a bit that indicates whether the corresponding range will be accessed by an offloaded block. Therefore, we need 97 bits to support a 48-bit virtual address space. Assuming the maximum number of entries is 100, we need 9,700 bits for the memory allocation table shared among all SMs. Third, each entry in the offloading metadata table contains begin/end PCs, bit vectors for live-in/live-out registers, a 2-bit tag for the bandwidth reduction on TX/RX channels, and the condition to offload it. We estimate this structure needs 258 bits based on CUDA PTX ISA 1.4 [41]. Assuming we need to support 40 entries in the table, which is 2× the maximum number of entries observed in our workloads, we need 10,320 bits for this structure in each SM.

We estimate our proposal's area with CACTI 6.5 [58] using the 40nm process node, the same process node our configuration assumes. The total area overhead is 0.11 mm^2, which is an insignificant 0.018% of the GPU area we model.

7. Related Work

To our knowledge, this is the first work that develops *programmer-transparent* methods for *offloading code* and *co-locating data and code* to enable near-data processing in GPU-based systems that employ 3D-stacked memory. No previous work we know of developed methods for automatically (1) identifying instruction blocks from unmodified programs to offload to NDP computation engines, and (2) mapping data to multiple memory stacks such that the data and the offloaded code that accesses it are co-located on the same memory stack. In this section, we discuss related works on 1) processing in memory and 2) placing data close to compute units, both of which are related to different aspects of our proposal.

Processing in Memory (PIM) has been explored to integrate computation units within memory (e.g., [13, 15, 19, 23, 28, 42, 43, 49–52, 54]). This idea has been revived with the emergence of 3D-stacked memory technology, and there have been been a number of more recent proposals that aim to improve the performance of data-intensive workloads using accelerators or compute units in 3D-stacked memory [2–5, 14, 16, 17, 32, 46, 55, 60, 62]. These works either 1) require *significant programmer effort* to map computation to PIM computation units, 2) do not consider the data mapping problem to multiple different computation units in different memory chips, or 3) are not applicable to GPU-based systems.

Among these works, the closest to ours is Lee et al.'s proposed compiler algorithm to automatically map code to a FlexRAM architecture [31]. This algorithm estimates the execution time of instruction modules on the host processor and the memory processor, and schedules instruction modules to the processor that would execute them faster. In contrast to our proposal, this work 1) is not applicable to modern GPU systems because the key bottleneck of GPU systems is memory bandwidth instead of latency and as such, 2) proposes a very different algorithm than ours, and 3) does not tackle the data mapping challenge to multiple memory stacks (as it assumes there is only a single memory chip with computation units).

Placing Data Close to Computation Units. The problem of placing data close to the computation unit that requires it has been examined in various contexts, including for NUMA (Non-Uniform Memory Access, e.g., [7]) and NUCA (Non-Uniform Cache Access, e.g., [25]) based systems. The challenge we solve is similar at a high-level, but our problem context (i.e., a GPU-based NDP system, which is inherently heterogeneous) is different and unique in two ways. First, as we discuss in Section 3.2, our data mapping mechanism needs to consider and satisfy the different requirements of both the NDP computation units and the main GPU for data mapping. For NDP units, data should be mapped close to computation units, but for the main GPU, data should be mapped to maximize bandwidth utilization. Past works did not tackle varying data placement requirements from heterogeneous execution elements. Second, in a GPU-based NDP system, there is no system software running on the NDP computation units: these units are controlled by the main GPU and their memory is pre-allocated.[10] As a result, dynamic and sophisticated data mapping mechanisms are less attractive in the GPU-based NDP system, and we develop a *simple data placement mechanism*, which no past work examined. Finally, none of these past works on data placement tackle the challenge of *automatically* finding code blocks to execute in near-data computation units.

8. Conclusion

We introduce TOM (Transparent Offloading and Mapping), a combination of new techniques that enable computation offloading and efficient data mapping to multiple 3D-stacked memories in a GPU system, without burdening the programmer. TOM consists of two key ideas. First, it statically identifies instruction blocks with maximum potential memory bandwidth savings from offloading, and dynamically decides whether or not the selected candidates should actually be offloaded based on runtime system conditions. Second, it exploits the predictability in memory access patterns of offloaded instruction blocks to co-locate offloaded code and

[10] In contrast, most previous works on placing data close to computation units in NUMA-based systems assume that system software can perform page allocation, migration, or duplication across the computation units.

data in the same memory stack. We have shown, through extensive evaluations, that TOM significantly improves performance while also reducing energy consumption and off-chip memory traffic, even with conservative assumptions about the internal bandwidth available in the memory stacks. We conclude that TOM is a practical and effective approach to enabling programmer-transparent near-data processing in GPU systems.

Acknowledgments

We thank the anonymous reviewers, members of NVIDIA Research and the SAFARI Research Group for their feedback. Special thanks to David W. Nellans, Daniel R. Johnson, Aamer Jaleel, and Evgeny Bolotin for feedback. We acknowledge the support of our industrial partners: Google, Intel, NVIDIA, Samsung and Seagate. This research was partially supported by NSF (grants 1212962, 1409723), ISTC-CC, SRC, DSSC, and the United States Department of Energy.

References

[1] N. Agarwal *et al.*, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *HPCA*, 2016.
[2] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.
[3] J. Ahn *et al.*, "PIM-Enabled Instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ISCA*, 2015.
[4] B. Akin *et al.*, "Data Reorganization in memory using 3D-stacked DRAM," in *ISCA*, 2015.
[5] O. O. Babarinsa and S. Idreos, "JAFAR: near-data processing for databases," in *SIGMOD*, 2015.
[6] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[7] R. Chandra *et al.*, "Scheduling and page migration for multiprocessor compute servers," in *ASPLOS*, 1994.
[8] K. Chandrasekar *et al.*, "System and circuit level power modeling of energy-efficient 3D-stacked wide I/O DRAMs," in *DATE*, 2013.
[9] N. Chatterjee *et al.*, "Managing DRAM latency divergence in irregular GPGPU applications," in *SC*, 2014.
[10] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[11] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*, 2004.
[12] R. G. Dreslinski *et al.*, "Centip3De: A 64-Core, 3D stacked near-threshold system," *IEEE Micro*, 2013.
[13] D. G. Elliott *et al.*, "Computational RAM: A Memory-SIMD hybrid and its application to DSP," in *CICC*, 1992.
[14] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA*, 2015.
[15] B. B. Fraguela *et al.*, "Programming the FlexRAM parallel intelligent memory system," in *PPoPP*, 2003.
[16] M. Gao *et al.*, "Practical near-data processing for in-memory analytics frameworks," in *PACT*, 2015.
[17] M. Gao and C. Kozyrakis, "HRL: efficient and flexible reconfigurable logic for near-data processing," in *HPCA*, 2016.
[18] M. Giles and S. Xiaoke, "Notes on using the NVIDIA 8800 GTX graphics card," https://people.maths.ox.ac.uk/gilesm/codes/libor_old/report.pdf.
[19] M. Gokhale *et al.*, "Processing in memory: The Terasys massively parallel PIM array," *IEEE Computer*, 1995.
[20] GPGPU-Sim, "GPGPU-Sim Manual."
[21] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.
[22] D. Johnson *et al.*, "Rigel: A 1,024-core single-chip accelerator architecture," *IEEE Micro*, 2011.
[23] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD*, 1999.
[24] J. H. Kelm *et al.*, "WAYPOINT: Scaling coherence to thousand-core architectures," in *PACT*, 2010.
[25] C. Kim *et al.*, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS*, 2002.
[26] D. H. Kim *et al.*, "3D-MAPS: 3D massively parallel processor with stacked memory," in *ISSCC*, 2012.
[27] G. Kim *et al.*, "Memory-centric system interconnect design with hybrid memory cubes," in *PACT*, 2013.
[28] P. M. Kogge, "EXECUBE-a new architecture for scaleable MPPs," in *ICPP*, 1994.
[29] D. U. Lee *et al.*, "A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *ISSCC*, 2014.
[30] D. Lee *et al.*, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.
[31] J. Lee *et al.*, "Automatically mapping code on an intelligent memory architecture," in *HPCA*, 2001.
[32] J. H. Lee *et al.*, "BSSync: Processing near memory for machine learning workloads with bounded staleness consistency models," in *PACT*, 2015.
[33] J. Leng *et al.*, "GPUWattch: enabling energy optimizations in GPGPUs," in *ISCA*, 2013.
[34] "Hybrid Memory Cube Specification 2.0," 2014.
[35] Micron Technology, "4Gb: x4, x8, x16 DDR3 SDRAM," 2011.
[36] D. J. Miller *et al.*, "Motivating future interconnects: a differential measurement analysis of PCI latency," in *ANCS*, 2009.
[37] V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.
[38] NVIDIA, "NVIDIA CUDA C Programming Guide."
[39] NVIDIA, "NVIDIA CUDA SDK 4.2."
[40] NVIDIA, "NVIDIA CUDA Toolkit."
[41] NVIDIA, "NVIDIA PTX ISA Version 1.4."
[42] M. Oskin *et al.*, "Active Pages: a computation model for intelligent memory," in *ISCA*, 1998.
[43] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, 1997.
[44] G. Pekhimenko *et al.*, "A case for toggle-aware compression for GPU systems," in *HPCA*, 2016.
[45] J. Power *et al.*, "Heterogeneous system coherence for integrated CPU-GPU systems," in *MICRO*, 2013.
[46] S. H. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *ISPASS*, 2014.
[47] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.
[48] T. G. Rogers *et al.*, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.
[49] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," in *MICRO*, 2015.
[50] V. Seshadri *et al.*, "Fast bulk bitwise AND and OR in DRAM," *CAL*, 2015.
[51] V. Seshadri *et al.*, "RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.
[52] D. E. Shaw *et al.*, "The NON-VON database machine: A brief overview," *IEEE Database Eng. Bull.*, 1981.
[53] I. Singh *et al.*, "Cache coherence for GPU architectures," in *HPCA*, 2013.
[54] H. S. Stone, "A logic-in-memory computer," *IEEE TC*, 1970.
[55] Z. Sura *et al.*, "Data access optimization in a processing-in-memory system," in *CF*, 2015.
[56] N. Vijaykumar *et al.*, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," in *ISCA*, 2015.
[57] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *MICRO*, 2010.
[58] S. J. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, 1996.
[59] D. H. Woo *et al.*, "POD: A 3D-integrated broad-purpose acceleration layer," *IEEE Micro*, 2008.
[60] D. P. Zhang *et al.*, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.
[61] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000.
[62] Q. Zhu *et al.*, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," in *HPEC*, 2013.
[63] W. K. Zuravleff and T. Robinson, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," 1997, US Patent 5,630,096.