PIMS: A Lightweight Processing-in-Memory Accelerator for Stencil Computations

Jie Li  
Texas Tech University  
jie.li@ttu.edu

Xi Wang  
Texas Tech University  
xi.wang@ttu.edu

Antonino Tumeo  
Pacific Northwest National Laboratory  
antonino.tumeo@pnnl.gov

Brody Williams  
Texas Tech University  
brody.williams@ttu.edu

John D. Leidel  
Tactical Computing Labs  
jleidel@tactcomplabs.com

Yong Chen  
Texas Tech University  
yong.chen@ttu.edu

ABSTRACT

Stencil computation is a classic computational kernel present in many high-performance scientific applications, like image processing and partial differential equation solvers (PDE). A stencil computation sweeps over a multi-dimensional grid and repeatedly updates values associated with points using the values from neighboring points. Stencil computations often employ large datasets that exceed cache capacity, leading to excessive accesses to the memory subsystem. As such, 3D stencil computations on large grid sizes are memory-bound.

In this paper we present PIMS, an in-memory accelerator for stencil computations. PIMS, implemented in the logic layer of a 3D-stacked memory, exploits the high bandwidth provided by through-silicon vias to reduce redundant memory traffic. Our comprehensive evaluation using three different grid sizes with six categories of orders indicate that the proposed architecture reduces 48.25% of data movement on average and obtains up to 65.55% of bank conflict reduction.

CCS CONCEPTS

• Computer systems organization → Special purpose systems;
• Hardware → Memory and dense storage.

KEYWORDS

Processing-in-memory; Hybrid Memory Cube; Stencil Computation; High Performance Computing

1 INTRODUCTION

The growing discrepancy between memory access time and the rate at which processors can generate memory requests, commonly known as the Memory Wall [31, 43], has limited the performance of modern computer systems. This is especially true for applications with low computational intensity that require large volumes of data. As memory bandwidth cannot be easily increased due to pin count limitations, the need for a large, fast memory puts greater pressure on the memory hierarchy. As a result, improvements to the memory hierarchy have become a critical area to explore to increase application performance. For massive datasets with limited temporal locality [20], the cache hierarchy does not work well and most accesses are served by main memory. This not only increases data access latency, but also leads to higher energy utilization.

Processing-in-memory (PIM) and near-data-processing (NDP) are different computing paradigms that can minimize data movement and reduce the corresponding energy overhead by integrating processing with main memory. PIM is not a new concept and a great deal of research was dedicated to the topic decades ago [19, 29]. Its limited success is due mainly to the associated design complexity, fabrication difficulty, and less immediate need. However, 3D integration technology now allows us to integrate logic and memory in a cost-effective way while the "big data" era is introducing massive datasets with a high demand for large memory bandwidth. Several PIM architectures and programming models have been proposed by academic projects [1, 2, 11, 38] and multiple memory vendors are starting to adopt 3D die stacking in mass-produced memory. These include JEDEC’s High Bandwidth Memory(HBM) [18] and the Hybrid Memory Cube(HMC) [4] proposed by Micron.

Stencil computation is a classic high-performance computing kernel. Stencil computations are generalized forms to solve systems of linear equations. Practically all applications in high-performance computing that model physical phenomena (weather, climate, physics, etc.) through finite difference or finite element methods employ rudimentary stencil operations. Stencil computations are often memory-bound as they present a high ratio of memory accesses to calculations and the overall dataset size for a multi-dimensional grid is typically much larger than the cache capacity of modern processors. Extensive efforts have been made to improve the performance of stencil computations. These improvements include leveraging tiling in both the spatial and temporal dimensions to increase data reuse within the cache hierarchy [24, 42] and offloading stencil computations to high-performance devices such as GPUs [25].

In this paper, we present the Processing-In-Memory accelerator for Stencil computations (PIMS), an implementation of the concept of processing-in-memory, specifically designed for stencil computations. This research is motivated by the fact that the benefits gained from the processor cache are limited. In a 3D stencil, we only gain a continuous benefit from cache size increases when the unit stride...
neighbors, the neighboring lines, the neighboring slices or the entire domain fits into the cache [36]. With typical cache and domain size, the neighboring slices may fit, but the domain is much larger than the cache on modern computing platforms. Elements beyond the cache have to be read from the main memory, leading to extra data movement and increased latency. The number of cache misses is extremely high if the stencil application is repeated hundreds of times. The PIM concept reduces off-chip transmissions and, thus, reduces the latency and energy overhead for these data-intensive applications. However, how to design a balanced and scalable PIM architecture is still an open question.

The contribution of this paper is three-fold: First we introduce PIMS, a novel PIM accelerator for stencil computations. PIMS is an extension of the Hybrid Memory Cube (HMC) device that recognizes stencil patterns and improves performance by processing the data which only need ADD operation. Second, we build a simulator generating the raw memory traces from the stencil kernel. The memory traces accelerated by PIMS are also produced from this simulator. Third, we evaluate PIMS with a variety of stencil patterns and provide a detailed analysis of the bandwidth utilization and performance improvements.

The rest of this paper is organized as follows. In Section 2, we briefly overview stencil computations and 3D-stacked memory, and discuss the motivation to design a PIM accelerator for stencil computation. In Section 3, we describe the proposed architecture in detail and discuss its trade-offs, including the request and response dispatcher, PIMS cache, PIMS operand buffer, computation unit and request builder. Section 4 provides the experimental evaluation and discusses the findings. Section 5 briefly surveys the related work. Finally, we present the conclusions and future directions of this work in Section 6.

2 BACKGROUND AND MOTIVATION

In this section, we provide the definition of stencil computation and an overview of 3D-stacked memory technologies. We then discuss how to exploit 3D-stacked memory to accelerate stencil computation.

2.1 Stencil Computation

Partial differential equation (PDE) solvers are ubiquitous in many scientific applications such as fluid dynamics, heat diffusion, and electromagnetics. These applications are implemented using iterative finite-difference techniques that sweep over a multi-dimension grid and perform nearest neighbor computations called *stencils* [5]. Each element in the grid is updated from a set of its nearest neighbors with weighted contributions in both time and space. For a given point in a 3-dimensional grid of order \(O\), the stencil can be defined as:

\[
U^{t+1}_{i,j,k} = c_0 V^t_{i,j,k} + \sum_{n=1}^{O/2} c_n (V^t_{i-n,j,k} + V^t_{i+n,j,k} + V^t_{i,j-n,k} + V^t_{i,j+n,k} + V^t_{i,j-k-n} + V^t_{i,j,k+n})
\]

(1)

In this equation, \(t\) refers to the time-step and \(c_n\) is the multiplicative coefficient applied to elements at distance \(n\) from the central point. \(U_{i,j,k}\) is the updated value, which will be used at the next time-step.

This kind of stencil requires \(O\) input elements in each dimension, not including the element at the intersection. Alternatively, the same stencil is also called a \((3 \times O + 1)\)-point stencil. We use ghost cells to store the boundary conditions. For a 3-dimension grid with \(\text{dim}_x \times \text{dim}_y \times \text{dim}_z\) grid size, the total grid size will be increased to \((\text{dim}_x + O) \times (\text{dim}_y + O) \times (\text{dim}_z + O)\). Even though space for these ghost elements could be saved by storing them separately (the eight \((O/2) \times (O/2) \times (O/2)\) corners of the array are not used), the savings are not significant enough to justify the increased code complexity. In this research, we consider the coefficient values to be constant scalars. They are only related to position and do not change with time.

To better understand the stencil, we represent a 6-order stencil with 19 points in Figure 1. For a grid size of \(\text{dim}_x \times \text{dim}_y \times \text{dim}_z\) of this stencil, we need another 6 ghost cells on each dimension to store boundary conditions. Figure 2 gives the pseudocode for this stencil kernel where the gray colors mark the neighbor elements which will be multiplied by the same coefficient. The coefficient values do not need to be continually read from memory for every new stencil. Instead, they can be hard-coded into the inner loop of the stencil code and kept in registers during the actual computation. This results in a reduction in potential memory traffic. However, the data structures where the stencils are stored are much larger than the CPU cache size. This leads to low reuse of data in the cache.

We simulate the cache performance of an 8-order, 25-point stencil with grid sizes of \(32^3\), \(48^3\), and \(64^3\) on 8-way, varying sizes of caches. As illustrated in Figure 3, when the entire stencil elements are larger than the cache size, the number of cache misses is very high. This situation gets even worse for larger grid sizes. If the cache is large enough to hold the entire domain, the number of cache misses would be reduced 85% on average and a larger cache would not provide any additional benefit. In our simulations, a \(64^3\) grid needs a cache larger than 1 MB to provide effective data reuse.

However, in many scientific applications that employ stencil computation kernels grid sizes are much larger than \(64^3\) and thus require correspondingly larger caches to exploit data locality. The L2 cache (256KB to 8MB) on modern processors is not large enough.
double *data-container, ***grid_a, ***grid_b, c[4];

/* Initialize data containers and coefficients */
data_a = malloc(sizeof(double) * (dim_x+6)*(dim_y+6)*(dim_z+6));
data_b = malloc(sizeof(double) * (dim_x+6)*(dim_y+6)*(dim_z+6));
init(data_a, data_b, c);

/* Generate contiguous layout of addresses */
grid_a = malloc(sizeof(double **) * (dim_x + 6) * (dim_y + 6));
grid_b = malloc(sizeof(double **) * (dim_x + 6) * (dim_y + 6));

/* Initialize stencil grids */
for(int i = 3; i < (dim_x + 3); i++) {
    for(int j = 3; j < (dim_y + 3); j++) {
        grid_a[i][j][k] = c[0] * grid_a[i][j][k] + grid_a[i-1][j][k] + grid_a[i+1][j][k]
        + grid_a[i][j-1][k] + grid_a[i][j+1][k] + grid_a[i][j][k-1] + grid_a[i][j][k+1]
        + grid_a[i][j][k+2] + grid_a[i][j][k+3] + grid_a[i][j][k-2] + grid_a[i][j][k-3]
        + grid_a[i-1][j][k] + grid_a[i+1][j][k] + grid_a[i][j-1][k] + grid_a[i][j+1][k]
        + grid_a[i][j][k-1] + grid_a[i][j][k+1] + grid_a[i][j][k+2] + grid_a[i][j][k+3]
        + grid_a[i][j][k-2] + grid_a[i][j][k-3];
    }
}
swap(grid_a, grid_b); // swap grid pointers
}

Figure 2: Pseudocode of 6-Order Stencil Computation

for these applications and will result in high cache miss rates. Obviously, the high cache miss rate degrades the stencil computation performance significantly, since a miss requires both accessing the main memory and handling the cache miss itself, which leads to a higher latency than just directly accessing the memory.

2.2 3D-stacked Memory

Through-silicon via (TSV) technology has enabled the manufacturing of a new class of memory devices based on the notion of stacking multiple DRAM and logic layers. This 3D technology not only improves the available bandwidth, it also reduces the latency of communication between circuits on different layers. High Bandwidth Memory (HBM) and Hybrid Memory Cube (HMC) are commercial devices using this technology that have shown significant potential toward improving the performance of memory-bound applications [31].

As shown in Figure 4, TSVs connect the various DRAM layers providing high bandwidth for data transactions within the memory. The logic layer implements the memory controller that manages the stacked DRAMs. Parallel high-speed links in the interposer layer connect the logic die to the host processor. In addition to the potential performance improvement realized by changing the fundamental physical memory organization and memory interconnect, many studies have started looking at opportunities to embed processing elements in the logic die of 3D-stacked memory with the objective to overcome the memory wall [43]. This in- or near-memory processing, often referred to as PIM operations, has the potential to reduce the host-to-memory bandwidth required to perform common operations such as atomic arithmetic operations, atomic boolean operations, and mutex locks.

However, the amount of logic actually embeddable in the logic layer of a 3D-stacked memory device is limited. The manufacturing processes for memory devices are not as refined as those used for high performance cores. Additional, more complex operations may lead to thermal issues. For these reasons, the logic die can be typically used to either accelerate specific instructions or to improve the actual memory management operations.

In this research, we use HMC as our memory technology. In HMC, the major organization unit is referred to as a vault. Each vault vertically spans each of the memory layers within the die using the through-silicon via (TSV). The vault logic block is analogous to a DIMM controller unit for each independent vault. These vault logic blocks and their respective vault storage units are organized into quad units. Each quad unit represents four vault units. Each quad
unit is loosely associated with the closest physical link block. All off-
band and in-band communication of HMC are packetized. Packets
specify single, complete operations. In this way, host processors
can minimize the latency through the logic layer of an HMC device
by logically sending request packets to links whose associated quad
unit is physically closest to the required vault.

2.3 Motivation
From Section 2.1 we can conclude that for every stencil point com-
putation, the majority of its operations are ADD operations that in-
volve up to six operands under the same coefficient. These operands
are the main sources of cache misses. In order to satisfy the high
bandwidth requirement of high-dimension stencil computations,
we propose moving the ADD operations under the same coefficient
inside the memory. The primary motivation for selecting this op-
eration for our PIM computations is that the hardware necessary
for the ADD operation is much simpler as compared to that of the
multiplication operation. This reduces the hardware area necessary
as well as the design complexity. Furthermore, reducing the data
movement for these operations provides a significant decrease in
memory access latency and power consumption.

The HMC 2.1 specification also provides the capability to perform
atomic operations in the logic layer. Currently, these atomic oper-
ations are limited to several basic operations that include bitwise,
boolean, comparison, and integer add operations. However, these
simple operations are sufficient to enable computation offloading at
the instruction level. As floating point add and subtract operations
are relatively simple compared to other operations, support for
these operations may become part of a future HMC specification.
Since most stencil computations employ floating point arithmetic,
in this work we also extend the HMC specification to support float-
ing point add operations.

3 ARCHITECTURE
There are several important factors to consider during the design of
a processing-in-memory accelerator. In this section, we first discuss
the design choices made and then present in detail the architecture
of our PIM accelerator.

3.1 Design Considerations
3.1.1 Offloading Target. The management of code offloading is
the first challenge to address in a PIM design [16]. In [1, 7, 8], the
proposed solutions require the programmers to identify the code
that should be run near to memory. This code is then supported
by the fully-fledged computational logic in their implementations.
However, as mentioned above, we utilize the computational logic
provided in HMC 2.1 instead of introducing new computational
units inside memory. Thus, the offloading target will be limited to
ADD operations. This provides us with multiple benefits compared
to prior approaches. Since the candidate code blocks for offloading
can be identified via static compile-time analysis, the complexity of
offloading code is minimized. Moreover, by moving relatively simple
computations next to the memory, the performance of computation
unit that performs the operation near memory will not be degraded
by executing compute-intensive instructions.

However, performance improvement for bandwidth-bound ap-
aplications is mainly obtained by reducing the off-chip memory
bandwidth consumption. It is therefore critical to choose as target
for offloading the operations that provide the largest bandwidth
reduction. As observed from Figure 2 in Section 2.1, most of the
stencil computations involved in one iteration of the inner loop are
the ADD operations under the same coefficient (blocks in gray). They
contain 15 ADD operations on 18 operands, consisting in 68% of
total operations and 78% of total operands respectively. Thus we
choose these operations as our offloading targets.

3.1.2 Programming Model. Figure 5 illustrates the programming
model used for stencil computations where the colored parts refer to
the data mapped to in-memory accelerators. The processing inside
memory is split into multiple smaller computations to enable con-
current processing. These smaller computations correspond to
the offloading granularity. After the PIM computations are complete,
the output data is sent back and processed by the host processor.
The host processor then processes the complex computations (multi-
lications) and non-data-intensive computing. No application-level
code change is necessary to enable PIM from the programmer’s
perspective. The only changes occur within the compiler, and do
not incur extra overhead for applications at runtime.

The PIM operations are expressed as specialized instructions of
the host processors. This provides the illusion that the PIM opera-
tions are executed as if they were host processor instructions [2].
We introduce a PIMload&add instruction that performs the compu-
tation on the operands under the same coefficient. We also adopt an
important design strategy here: for all the PIMload&add operations
under the same order level, the PIM accelerator only returns one
computed datum to the host processor.

3.1.3 Communication. Compared to proposals like [7], which use
DRAM memory-mapped registers for communication between pro-
cessors and PIM hardware, our implementation takes advantage
of the packet-based protocol of HMC to avoid the need for extra
hardware as well as redundant status checking communication. A
PIM accelerator is accessible to host processors via standard mem-
ory commands, while providing extended operations on the data

Figure 5: PIMS Programming Model
in memory. This concept has been employed for atomic operations in HMC 2.1. The packetized protocol allows each PIM accelerator to communicate with one another directly, without being coordinated by the host processor, which is orthogonal to previous works [7, 30]. The host processor can also communicate with PIM accelerators directly. In our design, the PIM accelerator does not issue any memory request, it only receives responses from other PIM accelerators. Instead, each memory request is generated by the host processor. Details will be discussed in Section 3.2.

3.1.4 Address Translation. In contrast to many other PIM architectures [11, 27, 38], our PIM accelerator does not support virtual memory addresses. This avoids the need for in-memory address translation, which may significantly increase access latency and requires area-inefficient translation units. Instead, when a host processor issues a PIMload&add instruction, just like other conventional instructions, it translates the virtual memory address of the target data element by accessing the TLB or walking through the page tables if a TLB miss occurs. The memory controller provides the PIM accelerators with physical addresses only.

This design strategy greatly reduces the complexity and improves the practicality of the proposed architecture since virtual to physical memory translation is managed on the host side with cooperation from the operating system. Pushing the virtual translation layer down to the memory device would require extra off-chip communication and would decrease the benefits gained from PIM. In addition, since our architecture does not utilize in-memory address translation, existing mechanisms for handling page faults do not need to be changed and all page faults can be handled on the host side. Furthermore, this methodology does not cause performance degradation to the TB. The PIMload&add instruction acts exactly the same as a standard memory request and requires only one TLB access [2].

3.1.5 Memory-side Cache and Cache Coherence. The DRAM operations in HMC follow a closed page policy: on completion of a memory reference, the sense amplifiers are precharged and the DRAM row is subsequently closed [12, 17, 27]. HMC reduces the row length (256B) to save power by alleviating the overfetch problem, where many unused bits are brought into the sense amplifiers. However, compared with the 8KB - 16KB rows in DDR3, shorter rows reduce the row buffer hit rate, making open page mode impractical [32, 34]. Furthermore, since each cube has many banks (512 banks in an 8GB HMC), always leaving the DRAM rows open will lead to high power consumption. The computation we offload to memory is data-intensive and may induce high bank conflicts without optimization. To reduce the number of accesses, bank conflicts, and repetitive data transactions in HMC, we introduce the PIM cache in the architecture. We refer to the memory-side cache as PIM cache in the rest of the paper, and discuss details of its design in Section 3.2.4.

Whether or not a given PIM solution supports cache coherence with the host processors has significant impacts on the complexity of the PIM implementation [22]. In our proposal, we isolate the CPU cache from the PIM cache to simplify the cache coherence mechanism. As such, there is no need to design mechanisms to enforce data coherence between the host processor and PIM accelerators. Since PIM accelerators have no access to the processor on-chip cache hierarchy, the data produced by the processor is not present in the PIM cache.

All PIMload&add memory request instructions bypass the CPU cache hierarchy and are directly offloaded to HMC, which integrates the PIM cache. All other memory requests from the host processor still go through the memory hierarchy where cache coherence is managed in the conventional way and bypass the PIM cache. Allowing PIM instructions to bypass the CPU cache provides extra performance benefits by avoiding cache checking overheads and reducing memory bandwidth. It also prevents cache pollution. It should be noted that for standard memory requests, bypassing the CPU cache would incur significant performance degradation.

3.2 Required Architectural Features

In this section, we describe our architecture that implements the PIM accelerator for Stencil computation (PIMS) with minimal modification to existing systems. Figure 6 gives an overview of our architecture. As mentioned before, the PIMS is built upon the fundamental structure of the HMC and is integrated into the logic die design while leaving intact the HMC interface; (3) it allows existing un-accelerated applications...
to run on PIMS-equipped platforms without incurring any performance penalty. Our architecture implements these features through four major components. First, we implement a Request/Response Dispatcher to distinguish PIMS requests from general load/store requests. It also delivers PIMS responses coming from other PIMS. Second, we introduce a PIMS Cache, which only buffers PIMS data, to exploit data locality and eliminate redundant DRAM accesses. Third, we add a PIMS Operand Buffer to orchestrate the computation sequence. Fourth, we implement a Response Builder to generate HMC specific packets. In the following sections, we present the work flow and explain these components in detail.

3.2.1 Workflow. For the non-offloaded data, both the request and response packets bypass the Request/Response Dispatcher and PIMS has no impact on these common load/store requests. We only discuss the workflow for requests offloaded to PIMS.

Figure 7 shows the execution path for PIMS requests: Step ①: the host processor issues a sequence of PIMS requests containing the corresponding physical addresses of the data under the same coefficient to the HMC and waits for the response. Step ②: once PIMS receives the PIMS request packet(s), it fetches and sends the data to the target PIMS indicated in the packet header and/or tail. Step ③: after PIMS receives all the needed data (all data under the same order level), it applies the floating point add operations and then builds the PIMS response packet and sends it back to the host processor. Step ④: the host processor receives the value returned from PIMS and the PIMS requests are satisfied.

3.2.2 PIMS Request Packets. We exploit the HMC header and tail packets in order to recognize the elements under the same order level and direct the data flow. Tag fields are used to accomplish the operational closure for requests. According to the HMC 2.1 specification [4], the tag fields in packets are eleven bits long. This allows representing up to 2048 pending transactions. As all tags are assigned and managed by the host, they will not be replicated in another request packet until a response with the original tag number is returned to the host. We slightly extend this protocol. For PIMS requests of elements under the same order level, the host assigns the same tag to these requests. Once a response with such a tag is returned, all these requests are considered executed and the tag is released.

3.2.3 Request & Response Dispatcher. PIMS only receives requests from the host processor. However, it may also receive responses returned from other PIMS. In order to separate requests from responses, we include the Request & Response Dispatcher in our PIMS architecture.

The dispatcher is transparent to standard requests and responses. All standard load/store requests will directly be sent to the vault controller. It is up to the vault controller to reorder its internal requests to optimize DRAM accesses and latency [4]. The responses also bypass the Request/Response Dispatcher and proceeds through the crossbar network.
PIMS requests are handled differently. For these, the Request/Response Dispatcher first checks if the requested data is in the PIMS Cache. If a cache hit occurs, the data is returned without any DRAM access. In contrast, a cache miss triggers a DRAM read using the appropriate cache line size. The dispatcher also checks the destination fields of the request packet. If the ID indicates the current vault, the dispatcher pushes the returned data to the PIMS Operand Buffer. Otherwise, it forwards the data to the crossbar switch. If an arriving packet is a response returned from other PIMS, indicating that the current PIMS is in charge of an ADD operation, the dispatcher pushes the data to the PIMS Operand Buffer for further processing.

3.2.4 PIMS Cache. As we mentioned in Section 3.1.5, the purpose of the in-memory cache is to minimize the DRAM accesses. However, we only cache the PIMS requests and bypass the standard requests based on the following consideration: for a PIMS-accelerated application, most memory requests are PIMS requests. In a 3-dimension stencil, PIMS requests are \( 30/(2 + 3.5O) \) of total requests, where \( O \) is the stencil order. In the 6-order, 19-point stencil, this percentage is up to 78.26% and will increase with higher orders to a theoretical maximum value of 85.71%. In addition, as standard memory requests are handled by the host side cache, the PIMS cache does not need to be coherent with the CPU cache.

HMC 2.1 supports request sizes from 32 to 256 bytes. In the event that the host processor issues a series of requests to the same block rather than issuing only one request accessing the entire data block, the probability of bank conflicts increases [40]. For example, if a HMC device has a maximum HMC block size of 256 bytes and the host issues 6 PIMS-load&add requests (i.e. 6 elements on the axis-x of a stencil grid) that fall into the same HMC block location, then the vault controller will send 6 independent requests to the specific bank. This causes the row in the bank to be opened and closed 6 times, significantly reducing bandwidth utilization and increasing latency. For this reason, adding the PIMS cache allows reducing the negative impact caused by consecutive accesses to the same block. If a cache miss occurs, an entire block of data will be copied into the cache, and the following requests to the same block can then fetch data directly from the cache itself.

3.2.5 PIMS Operand Buffer. The PIMS Operand Buffer is a hardware unit that temporarily stores the operands of pending ADD operations. The purpose of the operand buffer is to exploit the memory-level parallelism (MLP) [2] of the ADD operations. For operands in one computation, the operand buffer stores them in the same entry and orchestrates the computation sequence.

As depicted in Figure 10, we use a table to manage the data pushed into the PIMS Operand Buffer. For each arriving packet, PIMS first checks the tag in the packet against the tag field in the operand table. If a match is detected, the data will be pushed to the slot that the tag indicates. Otherwise, a new entry is allocated to store the data and the tag field is set. When the operand table is full, the following packets are stalled until space frees up in the buffer. PIMS monitors the data slot occupation. Once the occupied slots reaches the number of order level elements (which is 6 in a 3-dimension stencil) in one entry, the Valid field is set, indicating that the operands are ready to be sent to the computation unit. When the computation unit is available, this data will be flushed into the computation unit. Meanwhile, the response builder is informed that the upcoming result from the computation unit should be tagged with the corresponding tag value. After all data under the same entry is flushed out, the entry is freed for the following packets. The PIMS Operand Buffer implements a first ready first served (FRFS) mechanism. If multiple valid fields are set, PIMS flushes out the earliest one.

3.2.6 Computation Unit. As the HMC 2.1 hardware is not yet publicly available, it is not within the scope of this paper to discuss the implementation of the Computation Unit. In this research, we assume that it supports the single-instruction-multiple-data (SIMD) [41] paradigm that repeatedly performs an ADD operation on vector operands. The computation unit performs the ADD operation on one set of elements received from Operand Buffer in the same cycle and then sends the results to the Response Builder.

3.2.7 Response Builder. The Response Builder prepares the actual PIM-load&add response. We use two queues to hold the tags sent from the Operand Buffer and the data computed by the Computation Unit, as shown in Figure 11. Given that the Computation Unit processes data in order and the Operand Buffer sends out the tag and raw data at the same time, we can safely ensure that the tag corresponds to the output data. The Response Builder combines tag and data to build the PIM-load&add response packet and then sends the response back to the host processor.

4 EXPERIMENTAL EVALUATION

Since stencil computations typically are memory-bound, we focus our analysis on the improvements that PIMS provides to the memory subsystem. We do not directly measure improvements due to additional computational units.

4.1 Simulation environment

Our simulation environment consists of multiple components. We first implement a stencil kernel memory trace generator to produce the raw memory operations. We use a 32KB, 8-way cache simulator to simulate a processor cache and memory subsystem and generate the memory access traces, we consider a single-level cache with
4.2 Benchmarks and Parameters

To evaluate the performance of the proposed PIMS architecture, we select as benchmarks 3 different sizes of 3-dimensional stencil grids with orders varying from 2 to 12. They share the following properties: 1) the computation is performed in double precision; 2) they employ Jacobi iterations, whose computation involves reading one grid and writing one grid. The stencil grids are 64^3, 128^3, and 256^3, respectively, which are much larger than the cache capacity. Table 1 shows the configuration of our simulation infrastructure.

### Table 1: Simulation Environment Configurations

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Cache</td>
<td>8-way, 32KB</td>
</tr>
<tr>
<td>HMC</td>
<td>4-Links, 8GB, 256B-Block</td>
</tr>
<tr>
<td>Avg. HMC Access Latency</td>
<td>93ns</td>
</tr>
<tr>
<td>PIMS Cache (each)</td>
<td>fully associative, 8KB</td>
</tr>
<tr>
<td>PIMS Operand table</td>
<td>32 entries, 48B per entry</td>
</tr>
</tbody>
</table>

a relatively small size to avoid generating terabytes of memory trace files. We then implement the PIMS simulator, which models the behavior of the previously described components, taking as input the memory access trace. Our PIMS model then passes the generated HMC requests to HMCSim [21] to obtain data/responses from the simulated HMC devices. HMCSim also allows us to gather the internal statistics of the HMC devices. Once HMCSim provides the data/responses, a memory request is considered satisfied.

4.3 Results and Analysis

4.3.1 Memory Traffic. To better quantify the benefits of PIMS, we define a new metric, Memory Traffic, for measuring data movement. For stencil computation without acceleration, we calculate Memory Traffic (MT) as the product of cache misses and the cache line size(64B). The Memory Traffic of PIMS-accelerated stencil computation also includes the product of the number of PIMload&add responses with the size of a domain element(double precision, 8B) as defined in Equation 2. We also define the Memory Traffic Reduction (MTR) as shown in Equation 3:

\[
MT \ = \ \text{cache misses} \times 64B \ + \ \# \ \text{of} \ \text{PIMload&add responses} \times 8B
\]

\[
MTR = \frac{\text{Raw MT} - \text{PIMS MT}}{\text{Raw MT}}
\]

Figure 12 depicts the memory traffic reduction for all three grids with order varying from 2 to 12. We observe that the memory traffic reduction increases alongside the order and stays constant when the order reaches 8 for grid 256^3. However, it increases significantly for grid 256^3. For grids 64^3 and 128^3, the memory traffic reduction is approximately identical under the same order. We respectively see reductions of 34.61%, 42.32%, 46.20%, 48.75%, 49.55% and 49.97%, on average. The reduction is up to 72.07% and 72.57% for grid 256^3 with order 10 and 12, respectively. The average memory traffic reduction of 64^3, 128^3 and 256^3 is 46.23%, 44.23%, and 54.29%, respectively.

4.3.2 Bank Conflict. As mentioned above, data-intensive applications may generate a large number of bank conflicts without optimization. Given the closed-page memory architecture in HMC, it is not possible to coalesce requests in the memory controller [4]. Therefore, we use the PIMS cache to reduce potential bank conflicts. Figure 16 shows reductions of 14.21%, 25.91%, 34.42%, 44.44%, 51.43%, and 54.98% on average. The maximum bank conflict reduction experienced is up to 65.66% when using grid 256^3 with order 12. The bank conflict reduction pattern is similar to the pattern of memory traffic reduction, both of which have a higher reduction when the order reaches 8 for grid 256^3.

We also explore the Memory Traffic Per Stencil with/without PIMS. As shown in Figures 13 and 14, with PIMS-accelerated computing the memory traffic per stencil of all these grids is basically the same under the same order, and increases from 32 B/point to 112 B/point as the order increases. However, the memory traffic per stencil does not increase linearly for grid 256^3 without PIMS. It shows a larger increase at order 8 and reaches 409 B/point at order 12, which is almost 2X of the grid 128^3.

In order to discover what causes the extra increase for grid 256^3 without PIMS, we plot the cache misses in Figure 15, which shows that the number of cache misses for grid 256^3 grows nonlinearly when the order is greater than 8. This, in turn, causes a significant increase of the number of memory accesses. As observed in Figure 14, PIMS performs similarly across these grids under the same order. This explains why PIMS provides a higher memory traffic reduction for grid 256^3 with order 8, 10, and 12.
application maintain the same level across all the orders. In contrast, the bank conflicts without PIMS increase significantly with larger orders and reach approximately 107 millions in total at order 12. This is 2.9 times the number of bank conflicts in the PIMS-accelerated application. The bank conflicts for grid 128$^3$ with PIMS show the same characteristic as grid 256$^3$. As shown in 18, they are always at the same level. This implies that, with PIMS, larger order stencil computations do not incur extra access latency due to bank conflicts, as instead happens with non-PIMS-accelerated stencil kernels.

4.3.3 Bandwidth Utilization. The packetized protocol of HMC requires extra control information such as cube ID, address, tag, etc. as depicted in Figure 9. A HMC response packet carries not only the actual data, but also the associated control information. This control information requires 16 Bytes per packet (1 FLIT) regardless of the actual data payload. Figure 19 illustrates the bandwidth utilization of grid 256$^3$. Obviously, the bandwidth occupied by the data in the PIMS-accelerated application is smaller than the bandwidth of the non-PIMS application, especially when the order is high. The total bandwidth reduction is 20.8%, 39.65%, and 39.77% for orders 8, 10, and 12 respectively. However, the control overhead for PIMS increases with the order and reaches up to 1.74X of the data at order 12. The relatively high control overhead reduces the benefit provided by smaller data payloads. At order 2, 4, and 8 the total bandwidth of PIMS is 12.4%, 14.16%, and 14.40% higher than non-PIMS, respectively.
We refer to the Bandwidth Efficiency metric proposed in [40] as:

\[
\text{Bandwidth Efficiency} = \frac{\text{Request Size}}{\text{Request Size} + \text{Overhead}} \tag{4}
\]

Accordingly, the control overhead is \(1 - \text{Bandwidth Efficiency}\). As illustrated in Figure 20, the non-PIMS application achieves stable and high bandwidth efficiency, which is 80.00% across all orders. Conversely, the bandwidth efficiency for the PIMS-accelerated application is relatively low and decreases from 47.13% to 36.43% as the order increases.

In our simulation, every cache miss fetches 64B of data along with 16B of control overhead. This provides a bandwidth efficiency of 64/(64 + 16), i.e. 80.00% for the non-PIMS application. However, computing bandwidth utilization for the PIMS-accelerated application is more complex. We not only consider the standard cache misses, but also those requests that bypass the cache. For each set of \(\text{PIMload&add}\) requests that bypass the cache, PIMS loads 8B of data with 16B of control overhead, i.e. bandwidth efficiency is 33.33%. Although standard requests still go through the memory hierarchy, the bandwidth efficiency of these requests is the same as non-PIMS situation. As the order increases, the number of \(\text{PIMload&add}\) requests increase, such that the overall bandwidth efficiency decreases.

5 RELATED WORK

5.1 Stencil Computation

Many recent studies have focused on optimizing stencil computations on modern CPUs and GPUs [6, 14, 15, 23, 25, 33]. Micikevicius [25] proposed a GPU parallelization approach for 3D finite difference stencil computation that achieves an order of magnitude speedup over existing CPU implementations. Holewinski et al. [15] presented a code generation scheme for stencil computations on GPU accelerators, which optimizes the code by trading an increase in the computational workload for a decrease in the required global memory bandwidth. Maruyama et al. [23] proposed a compiler-based programming framework that automatically translates user-written structured grid code into a scalable parallel implementation for GPU equipped clusters. In another paper [33], Schäfer et al. investigated how stencil computations can be implemented on general purpose graphics processing units (GPGPUs). Dursun et al. [6] applied in-core optimization techniques such as cache blocking, register blocking, and software prefetching to improve high-order stencil computations. A domain-specific language and compiler for stencil computations was proposed in [14]. This allows specification of stencils in a concise manner and automates both locality and short-vector SIMD optimization along with effective utilization of multi-core parallelism.

Some efforts proposed novel algorithms to optimize stencil computations. Nguyen et al. [28] presented a 3.5D-blocking algorithm that performs 2.5D-spatial and temporal blocking of the input grid into on-chip memory for both CPUs and GPUs. Frigo et al. [9] proposed a cache oblivious algorithm for stencil computations that exploits temporal locality throughout the entire memory hierarchy. Strzodka et al. [37] developed a time skewing algorithm that breaks the memory wall for certain iterative stencil computations. Other research, such as [13, 35], explored the Execution-Cache-Memory (ECM) model to quantify the performance bottlenecks of stencil algorithms. Frumkin et al. [10] derived tight bounds on cache misses for evaluation of explicit stencil operators on rectangular grids.

5.2 Processing-In-Memory

There have been many efforts to leverage the infrastructure and protocols of the Hybrid Memory Cube to enable processing in memory [3, 26, 27, 30]. The Active Memory Cube [27] described a processing-in-memory architecture which implements a set of sophisticated computational elements on the logic layer below the stack of DRAM dies. Nai et al. [26] utilized the atomic operations specified in the HMC specification to enable PIM offloading without requiring additional programmer effort. The Smart Memory Cube [3] proposed by Azarkhish et al. presented a backward compatible and modular extension to the standard HMC which supports near memory computation on the logic base. Pugsley et al. [30] proposed a high-level description of the Near-Data Computing hardware and accompanying software architecture, which presents the programmer with a MapReduce-style programming model.

Other works implement processing-in-memory on GPU-based systems. Hsieh et al. [16] introduced a programmer-transparent architecture called TOM (Transparent Offloading and Mapping) which statically identifies instruction blocks with maximum potential memory bandwidth savings and exploits the predictability of in-memory access patterns to co-load offloaded code and data in the same memory stack. Zhang et al. presented a programmable, GPU-accelerated processing-in-memory architecture in [44]. Their approach exploited the throughput-oriented nature of GPUs to efficiently utilize the high bandwidth present between in-memory processors and memory dies and provided the programmability needed to support a broad range of applications. Some processing-in-memory accelerators designed for specific applications have also been discussed in recent studies. In [1], Ahn et al. presented a programmable PIM accelerator for large-scale graph processing. Gao et al. [11] developed hardware and software for a Near Data Processing architecture targeting in-memory analytic frameworks (MapReduce, graph processing, etc.). In the paper [2], Ahn et al. discuss how to employ compute-capable memory commands to implement in-memory computation, an approach similar to our own.
6 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed PIMS (Processing-In-Memory for Stencil Computations), a novel lightweight processing-in-memory accelerator targeted at improving the performance of stencil computations. It’s a fixed-functional PIM which has relatively simple logic, less power consumption and area size [39] than the programmable PIM approach such as [45]. PIMS recognizes offloading targets without extra programmer effort and offloads data-intensive operations to memory through PIMload&add instructions. It exploits the logic die of HMC devices to implement a request/response dispatcher, a PIMS cache, a PIMS operand buffer, and a response builder and enable stencil kernels acceleration. Our extensive evaluation results show that PIMS is able to significantly reduce redundant memory traffic up to 72.57% for large grids that exceeds the size of the host processor cache. The memory-side PIMS cache also maintains the number of bank conflicts constant as the order of the stencil computation increases, differently from conventional, non-accelerated, approaches. We also studied bandwidth utilization, identifying a reduction in utilization (39.77%) for stencils with high orders. We believe that PIMS is a promising approach to improve stencil computations with large sizes and high order datasets. Our current research focuses on two directions: 1. the study and development of mechanisms to improve bandwidth efficiency. 2. the investigation of the impacts of PIMS on latency and power consumption.

7 ACKNOWLEDGEMENT

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation under grant CNS-1358078, CNS-1362134, CCF-1409946, CCF-1718336, OAC-1835892, and CNS-1817094. We would also like to show our gratitude to the Graduate School of Texas Tech University for providing the Summer Thesis/Dissertation Research Award to show our gratitude to the Graduate School of Texas Tech University for the Summer Thesis/Dissertation Research Award.

REFERENCES

[19] Seth H Pugliesi, Jeffrey Jeste, Huizui Zhang, Rajeev Balasubramonian, Vijay-alakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC


