IMPRINT: In-Memory Processing with Indirect Addressing Techniques with GPU-hosted HBM-PIM

Ersin Cukurtas
ec9hc@virginia.edu
University of Virginia
Charlottesville, VA, USA

Kavish Ranawella
bue6zr@virginia.edu
University of Virginia
Charlottesville, VA, USA

Kevin Skadron
skadron@virginia.edu
University of Virginia
Charlottesville, VA, USA

Mircea Stan
mircea@virginia.edu
University of Virginia
Charlottesville, VA, USA

Abstract

Modern applications, like machine learning and graph/database algorithms, demand more memory capacity and bandwidth for efficient data access. This drives advances in memory technologies, including the shift from DIMM-based systems to high bandwidth memory (HBM) in some accelerators, as well as the concept of processing in memory (PIM), which moves computation to memory.

PIM boosts memory bandwidth, reduces energy usage, and preprocesses data. Recent industry products integrate PIM into HBM memory[13]. These PIM-enabled systems focus on a limited number of in-memory operations to avoid excessive overhead and we believe that enabling indirect addressing or virtual-to-physical address translation is a crucial one. Our design enhances GPUs with PIM-enabled HBM, adding indirect addressing through hardware and software co-design without requiring to perform page table walks. We achieve a general performance increase of $\sim 1.4 - 1.6x$ for large input sizes while providing $\sim 60 - 70\%$ energy savings with minimal area overheads.

Keywords

Processing-in-Memory (PIM), Sparse Matrix-Vector Multiplication (SpMV), High Bandwidth Memory (HBM), GPU, Max Pooling, Matrix Transposition, Memory Bandwidth, Page Table, Virtual Address

ACM Reference Format:
Ersin Cukurtas, Kavish Ranawella, Kevin Skadron, and Mircea Stan. 2025. IMPRINT: In-Memory Processing with Indirect Addressing Techniques with GPU-hosted HBM-PIM. In *International Symposium on Memory Systems (MemSys '25), October 07–08, 2025, Washington, VA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3767110.3767121

1 Introduction

The motivation for Processing-in-Memory (PIM) primarily aims to reduce data movement between the host processor and main memory, addressing the growing "Memory Wall" problem, which emerged in the 1990s due to CPU speed surpassing main memory speed [20]. Extensive research has attempted to tackle this issue through wide-issue architectures, better prefetching, and larger caches, but these solutions fall short for some data-intensive applications, including machine learning, databases, and graph algorithms [9].

Efforts to cope with the data volume include the development of new memory technologies, like High Bandwidth Memory (HBM), which leverages 3D stacking for higher bandwidth, lower latency, and reduced power consumption [12], [6]. HBM utilizes new packaging technology by 3D stacking multiple DRAM dies on top of each other connected with through silicon vias (TSVs) and moving the memory closer to the computation by placing it on the same System-on-Chip interposer as the host processor using 2.5D integration. Despite advancements in memory and interconnection technologies, the bandwidth challenge persists as processing power outpaces them.

PIM is an approach that mitigates bandwidth bottlenecks by performing computations directly within memory, reducing data transfer requirements and improving efficiency [23], [1], [2]. There are two PIM approaches: processing near memory and processing using memory [10]. This work aligns with the former, integrating PIM capabilities into HBM in conjunction with GPU as the host, following prior research [13].

Recent works have shown that even though PIM brings excellent performance and energy benefits, it is still difficult to beat GPUs in performance without extensive optimizations to PIM hardware. The first two implementations of Gearbox [15] were considerably slower compared to GPU implementations of the Gunrock benchmark [19]. The UPMEM PIM architecture shows great energy benefits compared to GPUs and can outperform GPUs for specific applications but still struggles for others [9]. TOP-PIM uses GPUs as in-memory processors because SIMD type GPUs are better at extracting bandwidth from memory due to the higher level of parallelism they can achieve without requiring complex hardware [23]. The most recent works that focus on accelerating SpMV through PIM[21], [3] also compare their results to GPUs and achieve considerably better results which will be discussed in Section 8. We see this as an indication that PIM should be used in conjunction with GPUs, because any kernel/application that is suitable for PIM would probably use a GPU as host and PIM can provide acceleration when memory bandwidth becomes an issue.

MemSys '25, Washington, VA, USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/10.1145/3767110.3767121

1.1 Target, motivation, and contributions

Given the advantages of GPUs and recent work showcasing GPUs with PIM-enabled HBM, this paper focuses on such architectures for applications especially heavy in pointer translation and indirection such as databases, key-value stores, graph processing, sparse matrices and less obviously matrix reshaping or transposition operations. Our motivation stems from the fact that given an architecture like our baseline architecture [13], the applications that can be run using PIM is limited and our goal is to enable indirect addressing by providing a simple way to translate virtual addresses to physical addresses in memory to extend the number of applications that can be run in PIM and showcase kernels that can take advantage of in-memory address translation.

Enabling in-memory address translation for PIM will improve performance where memory bandwidth becomes the bottleneck for applications that require indirect access through pointers. While previous work has studied address translation in memory [15], [11], [8], our approach shown in Figure 3, designed for GPU-hosted PIM-enabled memories, stands out for its simplicity as we do not perform page table walks. Our paper's contributions include:

- Propose enhancements to the target architecture [13] and develop a solution called *indirection engine* that enables a unique mechanism for virtual to physical address translation in memory with minimal area cost.
- Discuss a list of applications that have indirect addressing and show if they can benefit from the indirection engine.
- Provide analytical models for why performance increase is modest compared to some of the earlier works [21], [3] particularly for SpMV and whether or not they can be leveraged in the context of in-memory address translation.

2 Background

2.1 Baseline GPU + PIM-enabled HBM architecture

The presented architecture in [13] consists of an HBM-based GPU in which the memory is replaced with PIM-enabled HBM memory. We are using a modern a GPU in AMD's MI250 for our host. A table for comparison to NVIDIA's A100 is provided in Table 1. This is approximately our baseline architecture and the importance of this architecture stems from the fact that it does not require changes to the memory controller for launching PIM kernels and can work as regular memory if needed. In this work, we consider extensions to this GPU + PIM-enabled HBM architecture; specifically, we add a simple virtual-to-physical address translation mechanism in memory to support indirect addressing using the advantages of having a GPU as host.

2.2 HBM specific features

HBM memory differs from traditional DRAM by stacking multiple *DRAM dies* on top of a *logic layer* in the same package. The GPU+PIM enabled HBM architecture incorporates PIM components exclusively on DRAM dies, including PIM ALUs, an instruction table in CRF (Command Register File), a scalar register file to store constants in SRF and two general purpose registers per two banks as shown in Figure 1. Placing ALUs on the DRAM dies must be done carefully to avoid reducing memory capacity.

	MI250	A100
Architecture	CDNA	Ampere GA100
Process Node	6nm	7nm
Memory Bandwidth	3.2TB/s	2.0TB/s
FP16 Performance	383 TFLOPS	312 TFLOPS
TDP	500W	400W

Table 1: MI250 is slightly better than A100. We mainly focus on FP16 performance and all of our kernels/applications run on FP16.

This design offers the advantage of achieving higher bandwidth for PIM-based computations compared to a design where PIM logic is placed beyond the memory channel's data bus, potentially providing up to four times more bandwidth. Additionally, the baseline design supports *All Bank* and *All Bank PIM* commands, enabling commands to be received by every bank in a memory channel, which we leverage in our work.[1]

2.3 Upper bound on performance gain

It is essential to understand why the maximum performance of our baseline architecture is limited to $\sim 4x$. This is entirely related to the available bandwidth to the host versus maximum bandwidth achieved with the *All Bank* and *All Bank PIM* access. The external bandwidth of the memory provided to the host is fixed at 256 GB/s, whereas the internal bandwidth reaches 1 TB/s. Considering the HBM runs at 1 GHz with *tccd* = 4 cycles and 256 bit data interface from the banks to the ALUs, we get 8GB/s internal bandwidth for every two banks. With 256 banks per stack and 128 PIM ALU blocks, this is equivalent to 1TB/s internal bandwidth. So, whatever we do, we cannot exceed this upper limit for PIM acceleration. In other designs like [21], [3], the upper bound sits at $\sim 8x$. The parameters for our memory configuration are provided in Table 2 for comparison to other designs. Essentially, the number banks within a rank limits the speedup in this design. Please note that the hardware described in [14] mentions that the number of banks within a pseudo channel can determine the maximum speedup and mentions different numbers such as $\sim 16x$ and $\sim 8x$, but those are theoretical numbers when all the banks have PIM ALUs attached and *tccd* is not taken into account. The actual experimental memory has PIM ALUs between every two banks and with *tccd* = 4 cycles and that limits the maximum internal bandwidth to $\sim 4x$.

[1] We make the assumption that the baseline [13] has already addressed issues related to the power implications of All Bank commands as well as the associated timings (e.g., tFAW, tRRD).

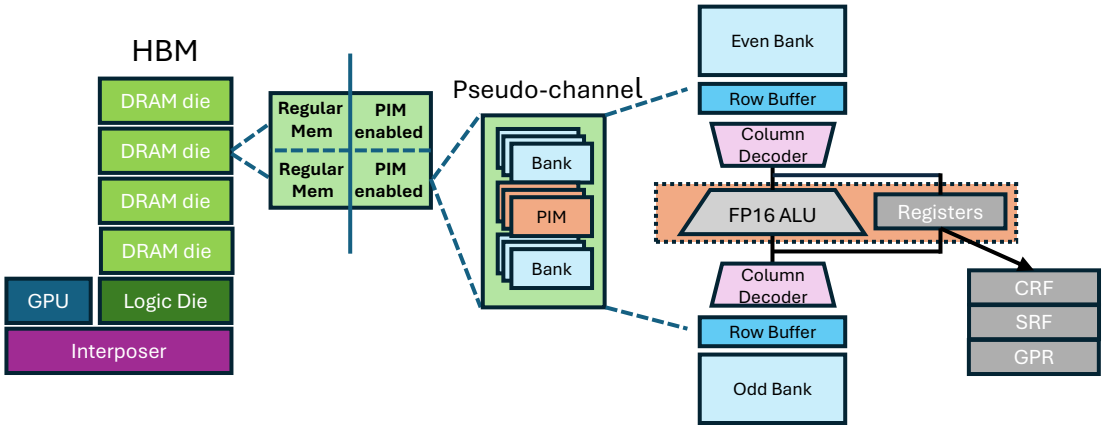

Figure 1: Baseline PIM architecture is based on [13]. Memory is divided into two. One half serves as regular memory (8GB) and the other is enhanced with PIM (4GB). This is to reduce the impact on memory capacity. Our focus is to enhance the PIM enabled portion further with the indirection engine.

Field	Value
Protocol	HBM2
# of bank groups per pCH	4
# of banks per group	4
# of memory rows	65536
# of memory columns	32
Width of the data bus	256
Row Buffer	8192-bit
# pseudo channels	16
Clock Frequency	1GHz
External/Internal Bandwidth	256GB/s, 1TB/s
Capacity	4GB

Table 2: Memory parameters used in our simulations on gem5

2.4 PIM-friendly GPU features

GPUs, in particular discrete GPUs which we assume in our baseline, have other features that more easily enable PIM designs we leverage. As a standalone device, the GPU driver is free to manage its own device memory. One key component is that page sizes can be more arbitrary than what is typically controlled by the operating system on a CPU. Larger page sizes are common in GPUs as a way to reduce TLB pressure and provide a way to guarantee a data structure is contiguous in both virtual and physical address space. This is important for PIM as it enables the broadcasting of PIM commands across multiple memory banks and channels over the address space.

Larger pages are also beneficial in the context of indirection. When performing translations, it provides simple arithmetic to perform translation as the physical addresses are guaranteed to be contiguous over a wide range and reduces the translation mappings that must be stored, similar to how larger pages reduce TLB pressure. Translation mappings on a standalone device are additionally set up before a GPU kernel is launched and can be more easily set up to not change during kernel execution.

2.5 Indirection

Indirection is a commonly used technique in software to represent data structures using pointers. An indirection occurs when reading a pointer from memory and using the value stored in that memory as an address to another location in memory. This process can be repeated multiple times until a non-pointer value is reached.

Pointers and indirection are useful in several situations in contemporary applications. The first case is representing sparse data. For example, machine learning matrices may be stored in a compressed sparse format rather than storing a matrix that contains mostly zero values. Graph representations of data used in graph algorithms are typically very unstructured, where a node in a graph may point to almost any other node in the graph, and as such, sparse matrix or linked-list formats are common here as well. Another case is representing a large amount of data where searching through the data needs to be done efficiently. Databases, for example, typically have an index consisting of pointers to traverse through a large amount of data.

One commonly studied area in PIM is accelerating pointer chasing in PIM [11]. The key insight with this acceleration is the latency needed to repeatedly dereference a pointer and return to the host becomes large for deep pointer traversals and is proportional to the number of pointers dereferenced. In this work, we focus on applications which contain many pointers with limited reuse.

2.6 Pointer layouts and PIM amenability

PIM typically excels in applications that have contiguous data. Although it may not seem useful in applications that make use of indirection, data pointers themselves sometimes are laid out in memory in a contiguous fashion. For example, compressed matrix formats will contain pointers in the form of offsets to non-zero elements, database indices contain a contiguous hash table that point to data elements determined by a hashing function, and graph formats such as adjacency matrix/lists can be laid out with a contiguous vector of all graph nodes followed by a linked-list of which nodes each is connected or represented as a sparse matrix.

Further, there are less obvious use cases, such as data layout transformations that take a contiguous array of memory as input, and output data which is redistributed in a different way. The output, in this case, can be viewed as an offset to another memory location which acts similar to a pointer. These types of transformations are common in ML applications which commonly perform operations to transpose a matrix, reshape or reduce data.

2.7 PIM terminology

Throughout this paper, we describe multiple ways of interacting with PIM and PIM-related components. These terms are defined here as they are used in this work:

PIM commands are the micro-operations which execute on the ALUs in PIM memory. PIM commands can be issued by the host, memory controller, or self-issued by memory. *Indirect PIM commands* are PIM commands which interact with the PIM features proposed in this work to support indirection.

PIM instructions are memory requests executed by the host GPU with PIM as the destination. These are typically store instructions with a payload containing the PIM command.

3 High-Level Ideas

Implementing indirect addressing support in PIM comes with several challenges. We list these challenges and explain how we overcame them in this section.

3.1 Light-weight address translation in PIM

Applications with indirect accesses to data structures store pointer values as *virtual addresses*, while PIM needs to issue requests to *physical addresses*. Therefore, some form of address translation is required before PIM can access the physical address. The obvious solutions for this address translation challenge are: (a) adding page walk logic to PIM, and/or (b) introducing additional translation look-aside buffers (TLBs) at PIM. Both performance and implementation considerations rule out a page walker implementation at PIM, whereas area and coherency (to keep TLB updated with the rest of the system) considerations make TLBs less attractive. To address this challenge, we propose a light-weight software-managed PIM address translation table (PTT) that leverages the fact that our host is a GPU and this provides friendly properties which can be summarized as: (1) We have large, flexible *physical* page sizes which means that contiguous virtual addresses can be known to be physically contiguous by looking at the page table, added to the fact that larger pages reduce the number of translation entries that need to be stored; and (2) It is guaranteed that virtual to physical address translation mappings will not change during kernel execution. This avoids problems related to translations being stale and removes the need for TLB-like "shoot downs" to be sent to the PTT.

Entry	Virtual Base	Physical Base	Size	Perms
0	7FFF'0020'0000h	3'F020'0000h	0x200000	RW
1	0h	0h	0	-
2	0h	0h	0	-
3	0h	0h	0	-

Figure 2: A populated PIM translation table (PTT). Virtual address is specified by software while the physical base, size, and permissions are populated into a PIM instruction generated by the GPU.

The PTT stores the minimum amount of information needed to perform the translation of an indirect address. This information includes virtual page number (VPN) to physical page number (PPN) translation, page size, and read/write permissions. The PTT is populated using *PIM instructions*, which are modified load/store instructions executed on the GPU. An example of a populated PTT is shown in Figure 2.

On the software side, the programmer or compiler specifies which data structures are used with indirect PIM commands. The base virtual addresses of these data structures are populated in the PTT. The PIM instruction populating the PTT will be translated the same as a regular GPU memory instruction. The GPU embeds the payload of these PIM instructions with a virtual address, physical address, page size, and permissions passed onto from the OS.

Based on our evaluation of several kernels for which implementation details are discussed in Section 5, the number of data structures tracked by the PTT in most kernels is one or two. That means the area overhead of this PTT is small and can be modeled as a 32 entry CAM. This small size also enables the PTT to use associative lookup when indirect PIM commands are executed.

3.2 Range and permissions checking

To prevent malicious applications from attempting to use the PTT feature to access data outside the range of the application, we must check that the translated physical address is within the range of the page size. Additionally, for our proposed architecture, we need to ensure that address is accessible by this PIM unit, i.e., the address must be within the same channel. The program must also have permission to read or write the page being indirected.

The range check is performed simultaneously with the address translation. If the result of the subtraction is negative, i.e., the virtual address is below the base virtual address of the PTT entry, it is out of bounds. Similarly, if the result of the subtraction is larger than the page size of the PTT entry, it is out of bounds.

The channel bits of the translated physical address must also be compared against the HBM channel ID to determine if the request is in the same channel. Requests that are outside of the channel will fail translation as IMPRINT supports channel local indirection only.

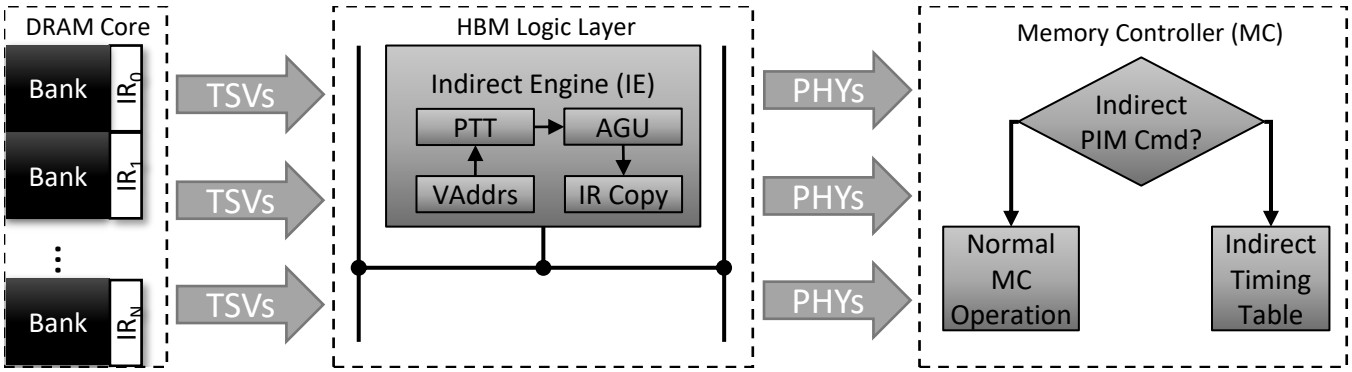

Figure 3: Diagram of memory side of proposed architecture. On the left the HBM adds logic to understand new PIM commands/operands and an *indirect register* (IR). In the middle an *indirect engine* is added to the HBM logic layer which includes an address generation unit (AGU) and PIM translation table (PTT) from Figure 2. On the right shows how the memory controller (MC) operates. The MC operates normally if a request is not an indirect PIM command and uses IMPRINT specific timings otherwise.

Permission bits are checked in parallel with the translation by comparing them against the access type (read or write) of the indirect PIM command. If sufficient permissions are not granted, the range check fails, or the channel local check fails and the translation fails.

3.3 Handling failures

Translation failures in our proposed architecture simply return an address of zero. Since there are many parallel translations, this allows kernels to continue on translations that were successful. Eventually, this may result in incorrect output of the application however, in our evaluation, we found that checking for failures in software after each indirect PIM command is too slow. Error handling is more easily enabled through hardware support. This support could be implemented as an error register populated with the first translation failure that can be checked by the program, an interrupt sent to the host, or similar mechanism and is a topic for future work.

Although translation failures seem inevitable, we describe later in Section 5 how kernels can be ported to PIM without failures. There we show how data structures are placed in memory to ensure that translation failures do not occur in any of the ported kernels.

3.4 Hardware-assisted data layout

Our proposed design requires data to reside within the same channel to perform indirection without translation failures. This is because allowing data access to any memory region, such as in another channel, would require coordination at the memory controller level to address memory in other channels. However, this extreme scenario would result in access patterns resembling those of GPU applications, thereby mitigating the benefits of implementing indirection in memory.

While it is possible to lay out data using a software-only approach, we propose offloading to the DMA engine for mostly security reasons. First, the software may not know the address mapping scheme used by the hardware and would not be able to craft memory addresses such that data is placed correctly. Second, this address computation may be difficult due to concepts like address hashing used in hardware, which would change the computation based on the physical address. Lastly, a data structure attempting to place data in a single channel may require a large allocation in order to place data in such a way that it is placed correctly in a single channel.

So, we provide a way for programs to place data in specific channels. We propose leveraging pre-existing direct memory access (DMA) engines used for CPU-to-GPU transfer in the GPU and augment them with support to write to specific channels. This augmented DMA engine can be used to duplicate data across channels during the CPU-to-GPU transfers and provide a different virtual address associated with each channel to the program that can be combined with offsets to access data.

4 IMPRINT Architecture

Figure 3 shows the overall architecture. Other than ISA enhancements to PIM supporting additional indirect PIM commands, an *indirect register* (IR) is also placed in the PIM unit of each bank. The IR is a register that can be written similarly to a normal PIM register. The IR is further connected directly to the row and column decoders of a DRAM bank. Indirect PIM commands use the IR to specify row and column addresses rather than using the address in a DRAM command allowing each bank to activate a different row number. This is combined with the All Bank support in [13] to parallelize indirect reads, which would otherwise require multiple DRAM commands. *This key operation provides the main performance benefit of IMPRINT as we can leverage the All Bank feature to reduce the total number of activation cycles that would be needed by a traditional host performing indirection.*

The IR only needs to contain pairs of row and column address values. We size the IR in this work to be able to hold the worst-case number of pairs needed for a single bank. Based on the number of row and column bits possible in HBM, this makes the size of the IR similar to adding an additional PIM register. Given that the area

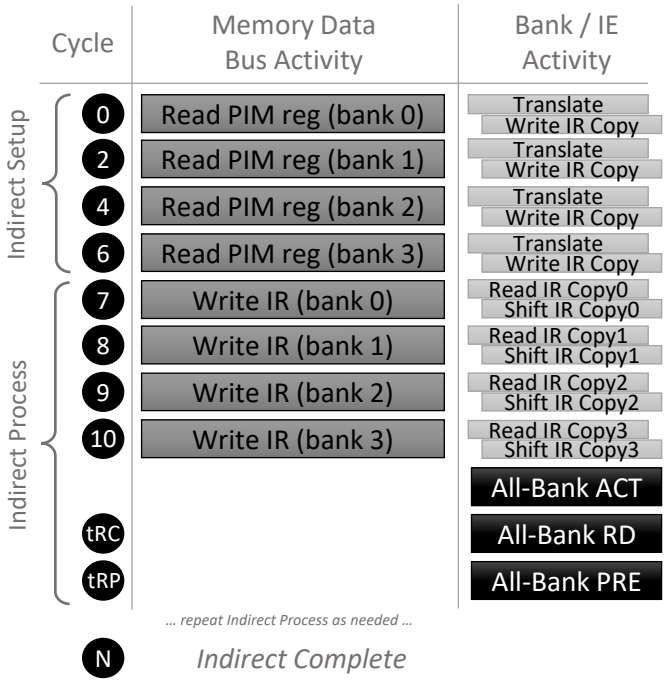

Figure 4: Example cycle-by-cycle diagram of an indirect PIM command shown on a 4 bank memory channel. Indirect PIM commands are segmented into indirect setup and indirect process commands.

is mostly dominated by ALU logic, adding a register costs little in area.

Indirect PIM commands direct all banks in a channel to use the IR to first perform an activate on itself. Next, the column bits in the IR are used to select which column to read. The column data is then shifted into a destination PIM register. For simplicity, in this work, we assume an All Bank precharge DRAM command is issued after each activate.

The more complex features of IMPRINT are placed in the HBM logic layer in our proposed *indirect engine* (IE). These features include translating indirect virtual addresses to physical addresses, bounds checking the dereferenced physical addresses, and writing the IR in each bank to trigger indirect reads.

4.1 PIM indirection flow

In a kernel employing PIM indirection, the process begins by populating the PTT with data structures requiring indirection support. Indirect PIM commands are issued and processed by the memory controller (MC) and the IE (Indirect Engine). The IE handles these commands in two steps: an *indirect setup* command, which configures IR copies for each bank, and an *indirect process* command, which triggers indirection and manages DRAM-related operations. This two-step process ensures predictable memory controller timings. The indirect setup command reads a PIM register specified in the indirect PIM command for all banks, with translations performed in parallel using an address generation unit (AGU) and the PTT. Afterward, the indirect process command writes the IR of each bank based on the values in each IR copy register from the setup command, enabling it to be divided into multiple commands for better memory request scheduling. The top portion of Figure 4 shows an example of an indirect setup in a memory channel with four banks and the bottom portion shows the indirect process.

Data read during indirect process commands is stored in a programmer specified PIM register until each bank's IR is empty. *The IE is responsible for organizing the data into the final PIM register in the order of input virtual addresses. If an IR register cannot complete all the translations in one iteration due to physical address pointing to a different bank, the unresolved addresses are buffered into a 32KB cache in the logic layer and redistributed for further translation through the IE.* A High-level view of PIM register values are detailed for SpMV in Section 5.

4.2 Overhead Analysis

Given the properties discussed in Section 3.1, we know that we will have a small number of entries in the PTT based on property (1) and that we will not have misses in the PTT based on property (2) and therefore, the PTT is basically a temporary storage space that can be modeled as a small 16 to 32 entry content addressable memory. The address generation unit either passes the base physical page number or adds an offset to it and therefore, this can be modeled as a 64-bit adder. We propose to add a register per ALU block with *access to row and column decoders* for self activation and another 32KB CAM for buffering unresolved address translations for redistribution through the logic layer.

The estimated area loss for the additional register even with a $4x$ adjustment is around 1%. Because the area around banks is largely dominated by the ALUs and we do not modify the ALUs in our design. The indirection engine and the 32KB CAM sits on the logic layer and we estimated entire area cost to be $0.483mm^2$. Given an average of $50mm^2$ die size, this would consume less than 1%. We use CACTI-3DD [5] for modeling memory blocks and standard circuit design tools for logic area estimation. Previous work have shown that the logic design in DRAM process could be up to $\sim 2x$ larger than the one designed in CMOS process due to limited number metal layers. So, everything is adjusted by $\sim 2x$ [22].

5 GPU to PIM kernel porting

5.1 Kernels suitable for PIM indirection

The IMPRINT architecture provides indirection support and enables indirection to occur in a more parallel fashion. As such, we expect to see the most benefit for applications with a large number of indirections.

Since our target is a PIM-enabled GPU, we also consider how the data structure layout may change compared to a CPU baseline. For example, a hash table may handle collisions on a CPU using a straightforward linked list. On a PIM-enabled GPU this data may be reorganized so that the linked list nodes are large enough to fit across multiple banks worth of PIM registers. As a result of this reorganization, much of the indirection classically seen on a CPU is gone, as the list nodes are now much larger.

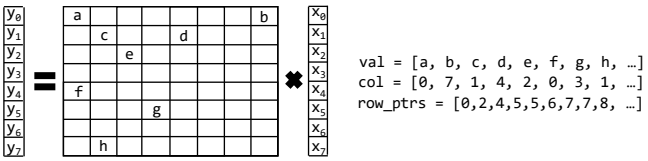

Figure 5: Example of compressed sparse row (CSR) used in Sparse Matrix-Vector multiply (SpMV). We compute y = A*x where the A matrix is represented as vectors: val, col, and row_ptrs. Entries in the matrix with no value are zero. val contains the non-zero matrix values which are in the column designated by the same entry in the col vector. row_ptrs defines the first element located in the row with that index.

Kernels we identified as having a large number of pointers even after reorganizing for a PIM-enabled GPU and that can take advantage of our parallel translation mechanism were sparse matrix processing (e.g., CSR) and data transformation algorithms such as matrix transposition, max pooling. All three types of kernels show similar performance improvements for large matrices. *The baseline for all experiments is GPU-HBM without PIM.*

5.2 SpMV

Sparse matrix-vector multiply (SpMV) is an application that uses a compressed matrix format to multiply with a vector and obtain a vector as a result. There are many compressed matrix formats, but we focus specifically on the *compressed sparse row* (CSR) format. All concepts applied to CSR apply to *compressed sparse column* (CSC) and other sparse formats as well.

An example of the CSR format is shown in Figure 5. The format contains the values of the matrix (the val vector), the column position of the value (the col vector), and row pointers which indicate the range of matrix values which are in the row corresponding to the index value of the row pointer.

The key observation for this application is that the val and col vectors are contiguous in memory. This makes the format potentially amenable to PIM processing as these vectors can be loaded into a PIM register using a basic copy command to move data from memory banks into PIM registers. However, the PIM register, which reads the col vector, does not contain the values needed but rather the *offset* into the vector we are multiplying matrix values with (the x vector).

At this point, the indirect PIM command is needed to convert the offsets into data from the x vector. The indirect PIM command will compute the virtual addresses using the offset and the PTT, perform an address translation, and read the values at that address into the PIM register. After this, we can continue PIM computation using normal PIM commands. The multiplication with the matrix values occurs next, followed by writing the output to a temporary buffer. Since the row pointers are difficult to work with in PIM, we choose to reduce these values using non-PIM GPU commands. Figure 6 shows the PIM portion of the GPU kernel along with the symbolic and numerical values in the PIM register after each command.

```
// Read columns into register 0.
PIM_read(cols, reg0);
```

col[0]	col[1]	col[2]	col[3]	col[4]	col[5]	col[6]	col[7]
0	7	1	4	2	0	3	1

```
// Operation to align 32 bit input to 64 bit lanes.
PIM_align(reg0, 4, 8);
```

col[0]	col[1]	col[2]	col[3]
0	7	1	4

```
// Dereference virtual addr in ind0 + offsets in reg0.
PIM_indirect_offset(reg0, ind0);
```

x[0]	x[7]	x[1]	x[4]
x_0	x_7	x_1	x_4

```
// Perform val[i] * x[col[i]].
PIM_mul_f32(reg0, val);
```

t[0]	t[1]	t[2]	t[3]

```
// Write result to PIM output vector t.
PIM_copy(reg0, t);
```

Figure 6: PIM portion of an SpMV kernel based on CSR format which uses PIM indirection. Column index values are first read and aligned. IMPRINT then reads the values of the *col* vector at the index value in the third PIM command. Column values are multiplied by the matrix values at the same index location and finally written to an output buffer for completion on the GPU.

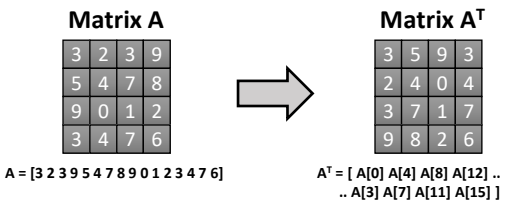

Figure 7: Example Matrix Transpose Operation. The arrays show the starting locations and desired output locations in memory.

5.3 Matrix transpose

The GPU implementation of matrix transpose is a well-studied problem, but the PIM implementation is usually not possible without specialized hardware in memory. In our architecture, where the processing happens at the bank level with registers and ALUs, matrix transposition is certainly not possible without our PIM indirection engine. As shown in Figure 7, the matrix *A* will be mapped as an array when copied over to the GPU memory, and the transposition problem is essentially relocation of numbers within this array. Therefore, it can be seen as an indirection problem which IMPRINT can accelerate. Since the new locations of each number within the

```
// Create offsets for A^T in CPU code.
Off = [0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15];

// Read offsets into register 0.
PIM_read(Off, reg0);
```

Off[0]	Off[1]	Off[2]	Off[3]	Off[4]	Off[5]	Off[6]	Off[7]
0	4	8	12	1	5	9	13

```
// Align 32 bit integer offsets to 64 bit
// lanes for pointer indirection.
PIM_align(reg0, 4, 8);
```

Off[0]	Off[1]	Off[2]	Offsets[3]
0	4	8	12

```
// Dereference virtual address in ind0 + offsets.
PIM_indirect_offset(reg0, ind0);
```

A[0]	A[4]	A[8]	A[12]
3	5	9	3

```
// Copy result to A^T
PIM_copy(reg0, A^T);
```

Figure 8: Matrix Transpose PIM Implementation. Output offsets are read into a PIM register, aligned, an indirect PIM command is issued, and the result is copied to an output buffer.

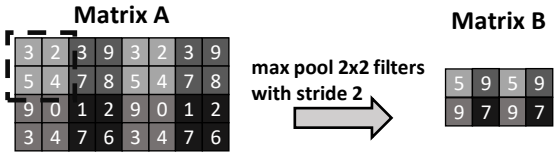

Figure 9: Max Pooling Operation. A small section of a matrix defined by a filter size is selected and reduced to a single element in the output by selecting the maximum value.

array are known beforehand, we can use these location "offsets" to load the correct numbers and form the transposed matrix. In Figure 8, starting with a *4x4* matrix, we can create an offset array for the transposed matrix and use these offsets to perform transposition in memory. Similar to the SpMV method, we load the first offsets to a PIM register and align them to load the numbers at their respective position within the array.

5.4 Max pooling

Max Pooling is a common operation in CNN algorithms used to reduce the dimensions and number of parameters of the feature maps after each convolution layer. Similar to the matrix transposition problem, max pooling is a matrix reshaping problem with a small function attached. In Figure 9, in order to compare values in a given window, first those values have to be retrieved from those locations and retrieving those values from "offsets" to the base pointer is where IMPRINT can help.

6 Evaluation

6.1 Methodology

We developed a model of our architecture in simulation using the gem5 simulator [4]. The most recent version of gem5 has the ability to run unmodified native GPU applications, which we use as our baseline applications. Our simulations use the KVM CPU model, which improves simulation time but introduces some non-determinism into the simulation. To help obtain better results, we perform multiple simulations of the same applications and average the run times of our test kernels to help obtain more deterministic statistics for our simulations.

Our model implements special gem5 pseudo-instructions on the GPU to model PIM commands. Our PIM-based simulated GPU issues PIM instructions containing an encoded PIM command which is executed by a PIM functional model. The PIM functional model contains all of the instructions demonstrated in the hardware prototype in [13] to start along with the new indirect PIM commands introduced in this work. We modify the memory hierarchy so PIM instructions arrive at the memory controller in the same order as the GPU executes them.

In addition, we developed a small gem5 specific runtime library that allows for PIM commands to be executed using simple function calls within the GPU programming model. Using this runtime library, the GPU kernels become essentially a list of PIM commands to be executed by the GPU. The runtime library also includes various helper functions for PIM, such as allocating memory in an aligned fashion, functions to initiate memory copies to specific channels using DMAs and address calculation functions to target specific channels with PIM commands.

To model the timing of the indirect commands, we implement the indirect setup and indirect process commands as described in Section 4.1. The indirect setup command takes a fixed latency defined by the number of banks in each memory channel. We then delay the next memory commands in the memory controller by a latency determined by the maximum number of indirections that must occur in a single bank. We simplify the scheduling logic by assuming the indirect process commands are issued, in full, directly after the indirect setup command. This avoids needing to develop a more complex scheduler, which is outside the scope of this work but is still realistic for the types of applications we are modeling that do not mix PIM and non-PIM memory requests at a fine granularity. Our simulator uses some of the elements described in [7]. We could not find a repository for this work, but we did utilize the Samsung PIM simulator repository for memory modeling [17].

6.2 Performance and Power

For Power evaluation we use the PIM Evalulation platform that has recently been published [18]. All the performance results are compared to a baseline of GPU-HBM with no PIM design.

SpMV: For SpMV, we first looked at how sparsity and matrix size affects the performance as shown in Figure 10. As it is obvious in other kernels as well, once the matrix size reaches a critical point,

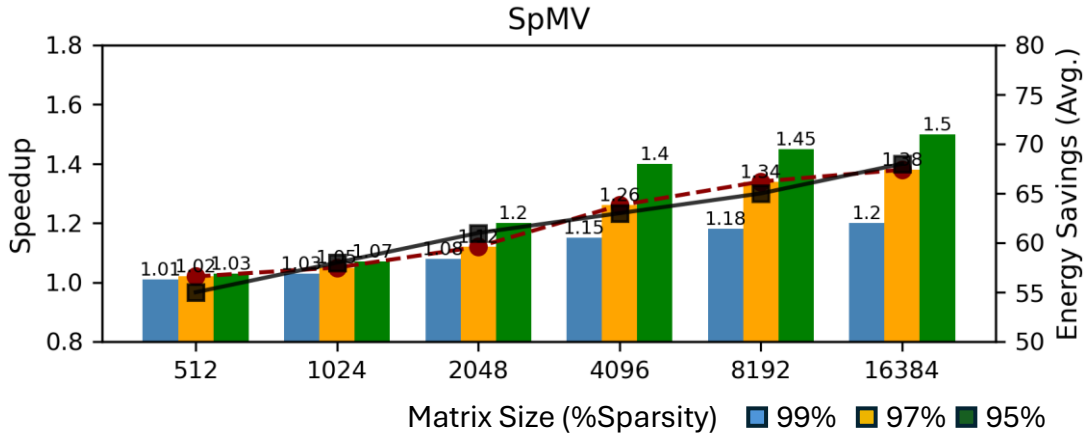

Figure 10: Both matrix sizes and sparsity affects performance. Basically below 97% sparsity and 4096 matrix size the resources are underutilized and the speedup seems to be capped around $\sim 1.5x$. Black line traces the average energy savings while the red line traces the average speedup.

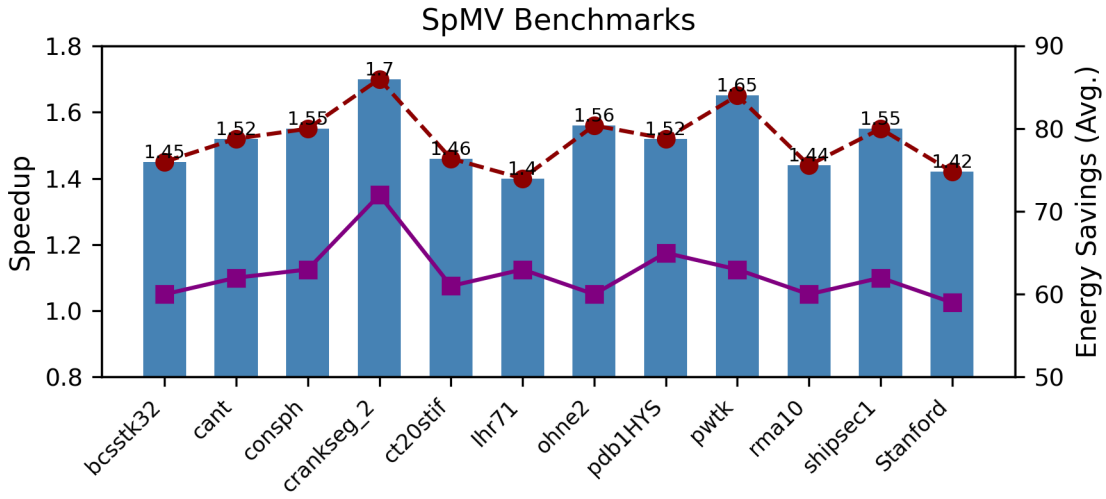

Figure 11: Benchmark evaluation for SpMV. Our design accelerates SpMV based on the number of non-zero elements. Purple line traces the average energy savings.

we are starting to get better speedups. And, the critical size in our design seems to be around 4096 by 4096. Below that number, we see an under utilization of resources and after that performance does not seem to go much higher. A similar relationship can be seen with sparsity levels. The less sparse the matrix is, the more utilization we get with our resources simply because, we perform more indirect access per iteration within IRs.

In Figure 11, we look at real benchmarks used in other works such as [21], [3]. From this figure, we conclude that our design accelerates SpMV based on the number of non-zero elements. All of these benchmarks are fixed to 99% sparsity, but they have more non-zero elements than the 95% sparse 4096 by 4096 matrix in Figure 10. Since we do not store any zero values and given the performance relationship to matrix size and sparsity, this conclusion makes sense. Also, the energy savings seems to follow the performance line. The downside of our algorithm is that the vector needs to be copied across channels, but we do not store *zero* values and so, this is negligible as opposed to storing most of the *zero* values as in [21], [3].

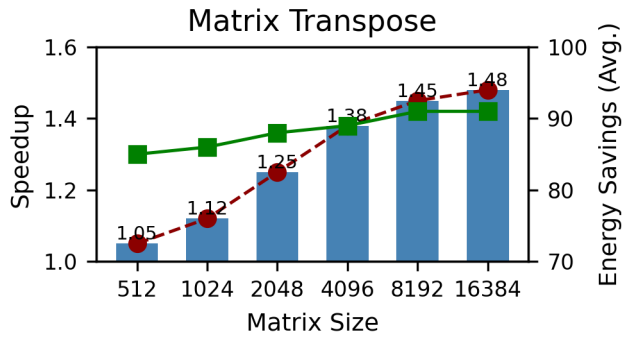

Figure 12: Matrix Transpose performance increases with matrix size. Green line traces the average energy savings

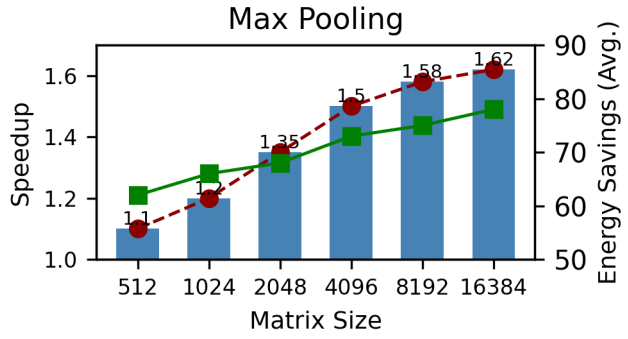

Figure 13: Max Pooling performance increases with matrix size. Green line traces the average energy savings

Matrix Transpose: We use similar matrix sizes as SpMV in matrix transpose based on commonly seen sizes in full machine learning applications. The performance improvement here has similar characteristics to SpMV, where the improvement is tied to how much of the resources are utilized. The sweet spot seems to be again around 4096 by 4096 as shown in Figure 12.

The energy reduction in this case is almost 90% when this kernel is executed alone, because no data is moved outside the memory. However, this is not realistic when we consider using this in applications like GPT-2. For instance, for the softmax function to be evaluated, we would need to send the transposed matrix to the host anyway for complex math operations like log or sqrt. This eliminates any potential benefit for energy reduction even though we could execute transposition slightly faster.

Also, through profiling we were able to find out that about $\sim 5-10\%$ is spent on transposition in GPT-2. This would mean that overall, we would get about 2% maximum speedup for such a large application which is minimal.

The downside of this solution is that the entire matrix needs to be copied across channels and executed in parallel and we have not yet found a large application where the benefits would outweigh the costs.

MaxPool: The MaxPool kernel is similar to matrix transpose in how it gathers data to be processed into PIM registers using IMPRINT. The kernel has an additional step of finding the maximum value in a PIM register which can easily be done in memory after the indirection step has been performed. As a result, the performance numbers seem to be very similar. Once again, this kernel sees improvement due to the IMPRINT feature and improves in performance with increasing matrix sizes as the resources start to get utilized as shown in Figure 13.

The energy reduction in this case is slightly lower compared to matrix transpose and it is following the performance numbers just like in SpMV. The reason for this is that, for both SpMV and Max Pooling, we still use the GPU for certain operations. For SpMV, the final reduction operation is performed on the GPU and for Max Pooling, we actually divide the pooling window across channels and to find the max value of the entire window, we gather the max value of smaller sections from the channels and find the final max value on the GPU. This ends up consuming more energy due to increased traffic to the host. But, with increased matrix sizes, the benefits of PIM outweigh the cost of increased traffic to host.

Just like in both SpMV and Matrix Transpose, some of the data have to be copied across the channels to actually execute this kernel with IMPRINT. The problem here is that the matrix has to be divided row or column wise into chunks and sent to different channels. However, when the pooling window in a single channel starts operating on the edge of each chunk, the window needs data from other channels and so, some of the data have to be duplicated to address this issue. For 3 by 3 window sizes, the duplication cost is around 16% per channel. But, with increased window sizes, this number would go up.

Unfortunately, Max Pooling only takes up $\sim 5\%$ of common CNN algorithms like AlexNET or ResNet. In fact, for all CNN algorithms, the execution time is dominated by matrix multiplication. So, the impact of accelerating Max Pooling for these applications would be minimal. However, there are other pooling methods, such as Diffpool, which are used in GNNs and this method can actually take up 30% of the total execution time. We believe that Diffpool could be similarly implemented using our mechanism and this could result in a net gain of $\sim 10\%$ in larger applications.

7 Discussion

7.1 Why is the performance increase modest?

We hypothesize that since with our architecture we do not utilize a network approach as in [21], our kernels reach a point with large matrices where they utilize all the PIM resources, but cross-bank communication becomes a bottleneck. Given the uniform distribution of data in our kernels, we can safely assume almost half the data have to travel through the logic layer for the next iteration of pointer translations. If half the data use the *external* bandwidth $\sim 1x$ as opposed to the larger *internal* bandwidth $\sim 4x$, then a simple calculation can be made to show that the maximum performance without a smart network on memory mechanism is limited to $\sim 1.6x$ in our architecture. That is $4/(0.5x1 + 0.5x4) = 1.6$. So, our maximum gain should be limited to $\sim 1.6x$. More importantly, even if our internal bandwidth was 32 times the external bandwidth, the acceleration would be around $32/(0.5x1) + (0.5x32) =\sim 2x$.

So, the speedup of this mechanism is limited to $2x$ by expensive cross-bank communication that goes through the logic layer. To overcome this bottleneck, we either would need to adopt a network-on-memory approach similar to SpaceA [21], or we would need to layout our data in a way that cross-bank communication is limited. Such methods are adopted in [3] and [8].

8 Related Work

Similar work has been done for memories with PIM capabilities [11], [8]. Even though they are providing a more general solution to virtual-to-physical address translation in memory, they are using TLBs and dealing with page table walkers whereas our solution is simply a software managed temporary storage space along with basic arithmetic units. This is because GPUs have features that can be leveraged to design a simpler solution specific to GPU hosts. Any algorithmic approach used in [8] can be leveraged to reduce cross-bank communication in our work and increase speedups as discussed in previous section.

The most recent works that focus on accelerating SpMV through PIM[21], [3] achieve a lot higher speedups compared to our work, but these works modify the memory architecture significantly and even though their architecture is somewhat different from ours especially in memory density. We estimate that overall they lose $\sim 10-20\%$ memory capacity as opposed to our $\sim 1-2\%$. Our work does not rely on a network-on-memory model. We believe such a model would improve the speedups to ideal case (maximum speedup) for our architecture as well and this is a topic of future research.

Gearbox [15] is another that lightly touches upon in memory address translation issue, but the goal of this work was to accelerate sparse matrix/vector operations in the Fulcrum architecture [16].Sparse matrix/vector operations inherently have indirect addressing due to representations of CSR/CSC formats, as discussed in Section 5.2. However, there is no indication that the described mechanism would work outside the range of the bank. Since pointer indirection was not the focus of the paper, the details were not shared. Our work solely focuses on pointer indirection, and we describe our indirection engine in detail in Section 4.

9 Conclusions

Our goal in this work was to extend the number of applications that can be run with PIM in a baseline GPU+PIM enabled HBM architecture with low overhead by enabling indirection support. The IMPRINT architecture provides the ability to perform indirection in a highly parallel fashion. This provides additional performance benefits by reducing memory cycles, minimizing bus bandwidth between host and memory, and enables a new wider range of pointer-based applications in PIM with minimal hardware cost.

Acknowledgments

This research has been funded by LPS (Laboratory for Physical Sciences) University of Maryland through the MIST Center. We appreciate their continued support and thank all the reviewers for their feedback towards improving this manuscript.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 336–348.

[3] Daehyeon Baek, Soojin Hwang, and Jaehyuk Huh. 2024. pSyncPIM: Partially Synchronous Execution of Sparse Matrix Operations for All-Bank PIM Architectures. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 354–367.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[5] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 33–38.

[6] Jin Hee Cho, Jihwan Kim, Woo Young Lee, Dong Uk Lee, Tae Kyun Kim, Heat Bit Park, Chunseok Jeong, Myeong-Jae Park, Seung Geun Baek, Seokwoo Choi, et al. 2018. A 1.2 V 64Gb 341GB/s HBM2 stacked DRAM with spiral point-to-point TSV structure and improved bank group data control. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 208–210.

[7] Derek Christ, Lukas Steiner, Matthias Jung, and Norbert Wehn. 2024. PIMSys: A Virtual Prototype for Processing in Memory. In *Proceedings of the International Symposium on Memory Systems*. 26–33.

[8] Amel Fatima, Sihang Liu, Korakit Seemakhupt, Rachata Ausavarungnirun, and Samira Khan. 2023. vPIM: Efficient Virtual Address Translation for Scalable Processing-in-Memory Architectures. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[9] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2021. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814* (2021).

[10] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 329–345.

[11] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. *2016 IEEE 34th International Conference on Computer Design (ICCD)* (2016), 25–32. doi:10.1109/iccd.2016.7753257

[12] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (high bandwidth memory) DRAM technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*. IEEE, 1–4.

[13] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, et al. 2022. Aquabolt-XL HBM2-PIM, LPDDR5-PIM with in-memory processing, and AXDIMM with acceleration buffer. *IEEE Micro* 42, 3 (2022), 20–30.

[14] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. 2021. Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 43–56.

[15] Marzieh Lenjani, Alif Ahmed, Mircea Stan, and Kevin Skadron. 2022. Gearbox: A case for supporting accumulation dispatching and hybrid partitioning in PIM-based accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 218–230.

[16] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R Stan, and Kevin Skadron. 2020. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 556–569.

[17] Seungwoo Seo Jin-seong kim Shin-haeng Kang, Sanghoon Cha. 2022. PIMSimulator. https://github.com/SAITPublic/PIMSimulator. Accessed: 2025-08-29.

[18] Farzana Ahmed Siddique, Deyuan Guo, Zhenxing Fan, Mohammadhosein Gholamrezaei, Morteza Baradaran, Alif Ahmed, Hugo Abbot, Kyle Durrer, Kumaresh Nandagopal, Ethan Ermovick, et al. 2024. Architectural Modeling and Benchmarking for Digital DRAM PIM. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 247–261.

[19] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on

the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[20] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[21] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[22] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. In-dram near-data approximate acceleration for gpus. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–14.

[23] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 85–98.