Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory

Bagus Hanindhito*
Ruihao Li*
Dimitrios Gourounas
hanindhito@bagus.my.id
liruihao@utexas.edu
dimitrisgrn@utexas.edu
The University of Texas at Austin
Austin, TX, USA

Arash Fathi
Karan Govil
Dimitar Trenev
arash.fathi@exxonmobil.com
karan.govil@exxonmobil.com
dimitar.trenev@exxonmobil.com
ExxonMobil Research and Engineering
Annandale, NJ, USA

Andreas Gerstlauer
Lizy K. John
gerstl@ece.utexas.edu
ljohn@ece.utexas.edu
The University of Texas at Austin
Austin, TX, USA

ABSTRACT

Wave simulations are used in many applications: medical imaging, oil and gas exploration, earthquake hazard mitigation, and defense systems, among others. Most of these applications require repeated solutions of the wave equation on supercomputers. Minimizing time to solution and energy consumption are very beneficial in this domain. Data movement overhead is one of the key bottlenecks that affect energy consumption.

Processing-in-Memory (PIM) has been a popular choice for deploying data-intensive applications, as a result of its high parallelism, low off-chip data movement and low energy consumption. In this paper, we propose an ISA-based, digital PIM system, to accelerate wave simulations. We fully explore the parallelism inside the algorithm, based on the size of the model and the availability of the PIM resources. We also examine the interconnections to optimize the inter-block data transfer. Our Wave-PIM can achieve an average of 41.98× speedup, as well as 12.66× energy savings, compared to the three state-of-the-art GPU platforms.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**.

ACM Reference Format:
Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy K. John. 2021. Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472512

*Both authors contributed equally to this research.

1 INTRODUCTION

Numerical solution of the wave equation is required in many applications. Acoustic and elastic wave simulations are used in oil and gas exploration [31], earthquake hazard mitigation [37], site characterization of critical components of civil infrastructure [18, 29], oceanography [14], medical imaging [22], and defense systems [25]. Electromagnetic wave simulations are used in modeling of antenna, radar, and satellites [45], design of electrical machines [5], defense systems [41], medical imaging [35], and oil and gas exploration [1].

The acoustic, elastic, and electromagnetic wave equation belong to the broader class of hyperbolic partial differential equations (PDEs) [39]. Hyperbolic PDEs share a lot of similarity when it comes to developing computational algorithms for their numerical solution[1]. They involve common operations[2], most of which are Level-1 BLAS, and the communication pattern is local.

In this work, we focus on the acoustic and elastic wave equation. We use strategies that make our work applicable to other hyperbolic PDEs with minimal adjustments. We use the discontinuous Galerkin (dG) method for the numerical solution of the equations, due to its accuracy, high data-locality, and ease of parallelization [24, 46]. Even though dG lends a lot of data locality to the computations, large models necessitate using distributed memory computing systems, which then entail inter-node[3] communication, in addition to intra-node communication. While many optimizations can be performed to limit data movement and overlap communication and computation, data movement remains among the key bottlenecks.

There are mature prior frameworks using multi-core CPUs to solve the wave equation [38, 46]. However, due to a considerable level of predictability, regularity, and parallelism in the algorithm,

[1]Another important member of hyperbolic PDEs is the Euler equation, with applications in aerodynamics, and aircraft/missile design, among others.
[2]These operations include: a) the problem domain is divided into many smaller elements (or cells), where each element has a number of nodes. The goal of a computational algorithm is to compute the solution field on these nodes; b) derivatives of certain fields need to be computed within each element. For tensor-product elements, such as cubes that we use here, the derivative computation involves a dot-product between a subset of the element's nodes, and a derivative vector; c) in order to compute the solution for the next time-step, certain scalar values (derivative values, and solution field from the current time-step) are combined to form updates, according to the utilized algorithm. Communication between immediate neighbors of an element is needed during this step; d) the updates are then combined with solution values of the current time-step to advance the solution forward for another time-step. This process (steps b-d) is repeated for as many time-steps as is needed.
[3]HPC node

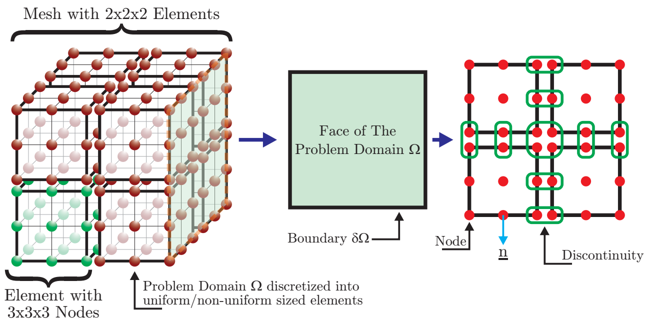
Mesh with 2x2x2 Elements
Face of The Problem Domain Ω
Boundary δΩ
Node
n
Discontinuity
Element with 3x3x3 Nodes
Problem Domain Ω discretized into uniform/non-uniform sized elements

Figure 1: The Discretization of Problem Domain using Discontinuous Galerkin Method

CPUs may not be the most efficient hardware to use. GPUs are alternative candidates, which can provide enough parallelism [4]. However, the scale of data sets exceeds the on-chip memory capacity of current GPUs [9, 10]. Due to the memory wall [47], the overhead of data movement between off-chip and on-chip memory quickly becomes the new bottleneck of the system if GPUs or other accelerators are used [26, 27]. In addition to limited on-chip memory, data movement introduced by the inter-element data synchronization inside each GPU is another key factor affecting the performance of the GPUs. Furthermore, the high energy consumption of GPUs is a key concern in data centers and supercomputers. As a result, a more efficient hardware solution with high performance and low energy consumption for wave simulation is desirable.

In this paper, we propose a Processing-in-Memory (PIM) system, which provides high parallelism while reducing the data movement cost, to speed up the wave simulation. We focus on acoustic and elastic wave equations. With minimal adjustments, strategies developed herein may be adopted in many other fields, such as major components of full-waveform inversion [16, 17], electromagnetic waves [6], and other hyperbolic partial differential equations [40].

Our system utilizes emerging resistive memories that have shown great potential for in-memory acceleration, since resistive memory cells can perform computation inside the crossbar logic in a parallel way [23, 44]. The mapping and layout techniques explored in this work are also applicable to other SRAM or DRAM based compute in memory technologies [15, 20]. We locate the hardware-level data locality in wave simulations and map it to an Instruction Set Architecture (ISA)-based PIM system. In addition, we further explore the inter-block data transfer inside the PIM architecture to improve the inter-element data synchronization. Specifically, our technical contributions in this paper include:

- We profile our GPU implementation of the acoustic wave simulation. Although the GPU solution can provide up to 369x speed-up over a high-end CPU, we find the main bottleneck is the off-chip data movement, which makes the simulation memory-bound, even with 900GB/s of off-chip HBM2 DRAM bandwidth.
- We present an architecture that improves the efficiency of the data movement inside the PIM.
- We layout the data in a hardware-friendly manner for the PIM architecture to minimize the overhead of inter-element data transfer, which is a key factor affecting the performance.
- We investigate H-tree and Bus interconnects to balance the trade-offs between the latency and energy consumption caused by the inter-element data transfer, and observe that the H-tree results in approximately 2.16× time savings in comparison to a bus architecture.
- We offer solutions to fold the workloads in batches or expand the workloads, to improve the scalability of the design, making it capable to support larger or smaller problem sizes at the highest possible performance.
- We evaluate our PIM design with extensive experiments on different PIM configurations. Compared to three state-of-the-art GPU platforms: GTX 1080Ti, Tesla P100, and Tesla V100, our PIM system yields an average performance increase of 45.31×, 34.52×, and 15.89×, and energy savings of 13.75×, 10.67×, and 5.66×, respectively.

2 BACKGROUND

2.1 A Prelude to Acoustic and Elastic Waves

The acoustic wave equation approximates the propagation of compressional waves in the Earth, water, and body tissue. It is described by the following PDE:

$$\frac{\partial p}{\partial t} + \kappa \nabla \cdot \mathbf{v} = 0, \tag{1a}$$

$$\frac{\partial \mathbf{v}}{\partial t} + \frac{1}{\rho} \nabla p = 0, \tag{1b}$$

where $p = p(x, y, z, t)$ is pressure, $\mathbf{v} = \mathbf{v}(x, y, z, t)$ is a vector that contains velocity values in the x, y, and z directions, denoted as $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$, respectively, κ and ρ are material properties, and $\nabla \cdot$ and ∇ are divergence and gradient operators, respectively.

The elastic wave equation describes propagation of compressional and shear waves in elastic solids, and is described by:

$$\frac{\partial \mathcal{S}}{\partial t} = \mu(\nabla \mathbf{v} + \nabla \mathbf{v}^T) + \lambda \nabla \cdot \mathbf{v}\, \mathcal{I}, \tag{2a}$$

$$\frac{\partial \mathbf{v}}{\partial t} = \frac{1}{\rho} \nabla \cdot \mathcal{S}, \tag{2b}$$

where $\mathcal{S}$ and $\mathcal{I}$ are the fourth-order stress and identity tensors, respectively, and λ and μ are material properties. One may observe structural similarities between Eq. (1), Eq. (2), and the Maxwell equations (not shown), which implies similar algorithms for computer simulations. In other words, successful strategies for efficient computation of the acoustic wave motion can also be applied to the elastic and electromagnetic waves. The acoustic wave equation considered in this work has four variables, while the elastic wave equation has nine variables that need to be evaluated at each mesh point in 3D space and at each time-step.

2.2 Problem Discretization Using the Discontinuous Galerkin Method

Figure 1 illustrates the discretization of the problem using the discontinuous Galerkin (dG) method, which enables higher data locality compared to alternative algorithms [46]. Our computational domain consists of hexahedral elements, each having their own

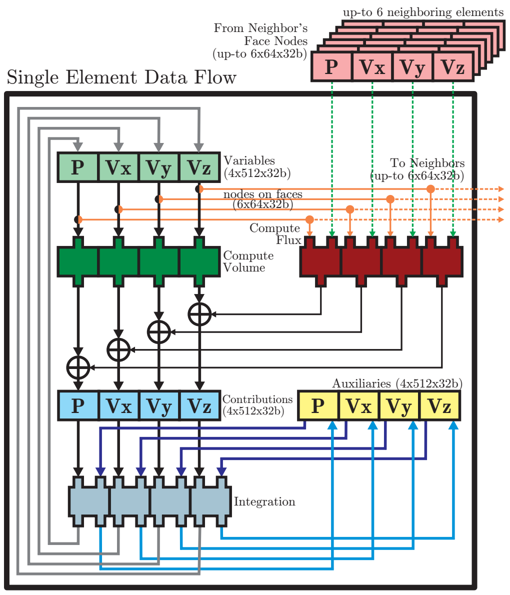
up-to 6 neighboring elements
From Neighbor's Face Nodes (up-to 6x64x32b)
Single Element Data Flow
P
Vx
Vy
Vz
Variables (4x512x32b)
To Neighbors (up-to 6x64x32b)
nodes on faces (6x64x32b)
Compute Flux
Compute Volume
Auxiliaries (4x512x32b)
Contributions (4x512x32b)
Integration

Figure 2: Single Element Data Flow Diagram

nodes. The dG algorithm computes the unknown values p and $\mathbf{v}$ at each node of the elements. Since each element has its own set of nodes, solution values across element interfaces can be discontinuous, as shown in the right part of Figure 1. The latter property enables the separation of computations into local and non-local groups, where each element has a generalized dataflow as shown in Figure 2.

Local computations only depend on nodal values within each element. These computations include: a) evaluating derivatives of the solution values (div $\mathbf{v}$, grad p), which has the form of a dot-product; and b) numerical integration of various values within an element, which has the form of a loop that sums up values. We group the two main local computations as *Volume* and *Integration*, which are drawn as green and grey blocks, respectively, in Figure 2. The compute *Volume* operates on variables to produce volume contributions. The *Integration* operates on (volume and flux) contributions to update the variables, and requires auxiliaries storage, all of which are updated during each integration step. There are five integration steps in each time-step.

Non-local computations reconcile discontinuous solution values at the interface of two elements. They need data values of corresponding interface nodes from a neighboring element (Figure 1), which is then used to compute *Flux* contributions. These non-local operations are drawn as a red block in Figure 2.

Increasing the number of nodes within an element improves solution accuracy. It also increases arithmetic intensity and the ratio of local to non-local computations. Table 1 summarizes key terms used in the dG discretization, which we refer to in this paper. The interested reader is referred to [24, 46] for further details.

2.3 Digital PIM Basics

PIM is an emerging hardware platform in which memory can perform as both compute and storage units. As a result, the data movement between processing units and memory units is eliminated, which is the bottleneck of most big data applications. Prior works have investigated both analog and digital characteristics of NVM to support arithmetic operations in memory. In analog PIM systems, dot product engines are achieved by applying different voltages to each wordline and bitline based on Kirchoff's Law [42, 43, 49]. However, the digital-to-analog (DAC) and analog-to-digital converter (ADC) blocks contribute the majority to the overall chip area and power, which can be avoided by exploring the digital characteristics inside the crossbar circuits [44].

In digital resistive PIM systems, arithmetic operations like addition and multiplication are achieved by performing NOR operations sequentially [23, 26, 44]. The NOR operation is achieved in memristor-based non-volatile memory (NVM) cells. The resistance of each memristor cell can switch between R_{ON} and R_{OFF}, which represents '1' and '0' in logic accordingly, when different voltages are applied to the bitline or wordline. The output memristor is initialized to R_{ON}. If one or more of the n inputs switches from '0' to '1', the output memristor will switch from R_{ON} to R_{OFF}. Due to the bit-by-bit NOR operations, the latency of the arithmetic operations on PIM may not be as efficient as other CMOS designs. However, the advantage of the PIM system is the high parallelism inside each memory block. As suggested in prior work [26], a 1GB PIM chip can support 8 million parallel operations.

3 DESIGN MOTIVATION

3.1 Bottlenecks in CPU/GPU Implementations

Our CPU code uses p4est [28] for mesh generation and workload distribution on multiple CPUs. It takes significant amount of time to run even a small-sized problem on high-end processors. GPUs could run wave simulations more efficiently than CPUs, due to the vast amount of parallelism inherent in the problem, both at the element- and node-level.

Table 1: Description of Terms Used in dG Discretization

Mass Inverse	The inverse of a diagonal mass matrix, which is constant, and is used repeatedly.
Unknown variables	Pressure (p) and velocity (v) values, which need to be computed at each node within an element.
Contributions	Incremental updates to unknown variables, which are computed during *Volume* and *Flux* computations.
Auxiliaries	Temporary storage for unknown variables needed during the temporal integration (*Integration*) step.
GLL Weight	Constant values for Gauss-Legendre-Lobatto weights.
GLL Point	Constant values for Gauss-Legendre-Lobatto points.
jacobian_det_w_star	Constant variable used in *Volume* integration.
jacobian_det_domain	Constant variable used in *Volume* integration.
jacobian_inverse_domain	Constant variable used in *Volume* integration.
jacobian_det_boundary	Constant variable used in *Flux* integration.
dshape	Derivative values of the shape functions.
κ	Constant material property: Bulk Modulus.
ρ	Constant material property: Density.
grad p	Gradient of pressure (∇p).
div $\mathbf{v}$	Divergence of velocity vector ($\nabla \cdot \mathbf{v}$).
λ and μ	Constant Elastic material properties.
grad $\mathbf{v}$	Gradient of velocity vector ($\nabla \mathbf{v}$).
div $\mathcal{S}$	Divergence of stress tensor ($\nabla \cdot \mathcal{S}$).
Refinement Level	Refinement Level n indicates the problem domain is discretized into $(2^n)^3$ elements.

Our GPU implementation of wave simulation, using CUDA, gives a significant speed-up over the CPU implementation, for a problem size that fits into the memory of a single GPU. In our experiments with GPUs, for mesh refinement level 4, with 1024 time-steps, a GTX 1080Ti, Tesla P100, and Tesla V100, reach speed-ups of 94.35x, 100.25x, and 123.38x, respectively, over a CPU implementation that runs on dual Intel Xeon Platinum 8160 with a total of 48 cores. For mesh refinement level 5, with 1024 time-steps, the speed-up over the CPU is 131.10x, 223.95x, and 369.05x, for GTX 1080Ti, Tesla P100, and Tesla V100, respectively.

In the above simulations, GPUs have not reached their peak performance. Upon profiling the simulator with `nvprof` [7], the GPU implementation of the acoustic wave simulation turns out to be bounded by memory bandwidth, even for Tesla V100 GPUs, with 900GB/s of memory bandwidth. This excessive data movement not only limits the performance, but also costs energy that can be higher than the computation energy itself. As noted in prior work, the energy cost of moving 256 bits of data by 10mm is significantly higher than the energy cost of a double-precision fused-multiply add [30].

The compute *Volume* kernel can benefit from more Streaming Multiprocessors (SMs), as we move from GTX 1080Ti, to Tesla P100, to Tesla V100, until the memory bandwidth becomes the bottleneck. Meanwhile, the *Integration* kernel does not scale so well, compared to the compute *Volume* kernel, since the memory accesses dominates this kernel. Lastly, the compute *Flux* kernel is the most inefficient kernel, since it has a large divergence that degrades the parallelism in GPU. Therefore, a solution that can process these computations in memory is desirable to reduce the data transfer cost.

3.2 Bottlenecks in Current PIM Architectures

Processing-in-Memory can reduce data movement to processors, however, dataflow between multiple kernels need to be handled efficiently. In prior works [26, 33, 43], PIM solutions achieved better performance than GPUs due to the removal of data movement between off-chip and on-chip memory. However, the internal data movement (from one resistive memory subarray to another) also affects the overall performance. In [26], the FloatPIM solution can achieve a 6.88× performance improvement if parallelized neighbor-block data transfer is enabled. But non-neighbor communication was not fully explored in FloatPIM, making it not a general solution.

Other prior works [33, 43] considered an identical latency for all inter-block transmissions, using a global buffer for data routing. The global buffer design is inefficient for transfer of adjacent blocks. Thus there is opportunity to improve inter-block data transfers within PIMs, while still exploiting near-neighbor links.

4 THE WAVE-PIM ARCHITECTURE

We now introduce the proposed Wave-PIM architecture, which incorporates H-tree based efficient interconnects for inter-block data transfers. Furthermore, we discuss how look-up tables can be implemented within the PIM architecture, due to their wide usage in High Performance Computing (HPC) applications [21, 36].

4.1 Overview

The Wave-PIM architecture is a digital PIM, where each PIM chip consists of multiple memory tiles, and each tile has multiple memory blocks as illustrated in Figure 3. The memory block is the most basic unit, which contains memristor memory cells, sense amplifiers, decoders, row and column drivers, and row and column buffers, as in prior work [26, 27]. Each memory block is identical, and computations are performed inside the blocks in a bit-serial way utilizing NOR operations inherently, without any separate ALU hardware.

Wave simulation can be abstracted as general memory instructions and arithmetic instructions. Instructions are sent from the host, and are pre-processed by the decoder on the PIM chip. Next, micro sequences are generated and sent to each memory block. Before the computation starts, we first issue memory copy instructions to load input variables to each memory block. When all required operands are ready, arithmetic instructions will be issued, and computations are performed inside memristor cells in a row-parallel way. Outputs are generated in the same row, and one additional batch of memory copy instructions will be issued for storing them.

4.2 Inter-block Interconnection

Arithmetic operations are processed inside each memory block, and the operands can come from other blocks, which makes the inter-block interconnection an essential component of the PIM system. We propose an H-tree-based topology to support inter-block data transmission. We also introduce how the H-tree-based topology can be extended to a Bus-based solution for better flexibility.

4.2.1 ***H-Tree.*** H-tree is a data structure widely used in VLSI, database systems, and DRAM interconnections [12, 13, 32]. Here, we demonstrate how to apply an H-tree in the PIM inter-block interconnect. The top-right part of Figure 3 gives an example of the H-tree interconnection in a 16-block memory tile. In an H-tree, each non-leaf tree node has four children nodes. Memory blocks represent leaf nodes, since a memory block is the lowest-level unit in our proposed PIM architecture. Therefore, there are 4 tree nodes (S_0), each controlling 4 leaf nodes (memory blocks), and 1 non-leaf node (S_1) controlling the 4 S_0 in each memory tile.

To show the detailed inter-block data transmission procedure, we consider an example of transferring data from Block 0 to Block 5. Before transmitting data, one read instruction (I_0) is sent to Block 0 first, for loading data from memristor cells to the row/column buffer. Then, `memcpy` instructions (I_1, I_2, I_3) will be sent to the H-tree nodes sequentially, along with the data path $D_0 \rightarrow D_1 \rightarrow D_2 \rightarrow D_3$. Finally, one write instruction (I_4) is sent to Block 5 to store the data from the row/column buffer to the assigned memory cells.

With the H-tree topology, the overhead of inter-block data transmission latency can be lower than the latency of prior PIM works [33]. For example, if the number of nodes in one element exceeds the number of rows/columns in the PIM memory block, multiple blocks can be used to complete the computation of one element. Since the blocks are under the same H-tree node, the data will only pass through one S_0 H-tree switch. Additionally, the number of children of a tree node does not have to be 4; it can be higher when customizing PIM systems for larger-scale models. For smaller-scale models

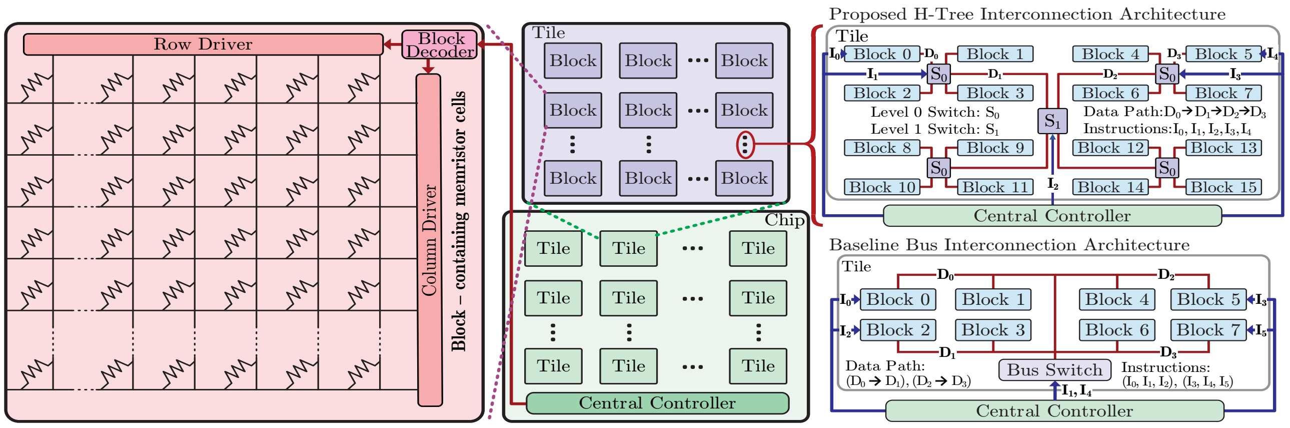
Row Driver
Block Decoder
Column Driver
Block – containing memristor cells
Tile
Block
Chip
Tile
Central Controller
Proposed H-Tree Interconnection Architecture
Block 0 … Block 15
Level 0 Switch: S0
Level 1 Switch: S1
Data Path: D0→D1→D2→D3
Instructions: I0, I1, I2, I3, I4
Central Controller
Baseline Bus Interconnection Architecture
Bus Switch
Data Path: (D0→D1), (D2→D3)
Instructions: (I0, I1, I2), (I3, I4, I5)
I1, I4
Central Controller

Figure 3: Wave-PIM Architecture Overview

when there is little inter-block data transmission, the low-overhead Bus interconnection can be used instead.

4.2.2 ***Bus***. The Bus interconnection can be an alternative candidate for inter-block data transmission, since the H-tree topology is inefficient, due to large leakage power, if there is little inter-block data movement. For example, in a 256-block memory tile, $1 + 4 + 16 + 64 = 85$ H-tree node switches have to be used. By contrast, only one central bus switch is needed for a bus interconnect.

The right bottom part of Figure 3 shows an example of transferring data from Block 0 to 2, and from Block 5 to 7. At first, two read instructions (I_0, I_3) are issued at Block 0 and 5 for loading data from memristor cells to the row/column buffers. Then, `memcpy` instructions (I_1, I_4) are sent to the bus switch sequentially, since only one data path can be enabled when using the bus interconnection. Finally, two write instructions (I_2, I_5) are issued at Block 2 and 7, to finish this data transfer.

Unlike the H-tree design, in which the transmissions between Block 0 to 2, and between Block 5 to 7, can be processed simultaneously, the bus switch processes these transmissions sequentially. This implies that if there is intensive inter-block data transmission, the Bus interconnection is less efficient than the H-tree. Trade-offs between these two interconnections will be discussed in the evaluation part (Section 7).

4.3 Look-up Table in PIM

Look-up tables are a commonly-used architecture in GPUs, FPGAs, and ASICs, since they allow replacement of complicated run-time computations with simpler array-indexing operations. In the digital PIM architecture, complicated arithmetic operations are implemented by Taylor series, or other approximate computation techniques [33, 48]. In addition to the arithmetic operations, complicated logic operations like generating the index of neighbor elements cannot be implemented with NOR operations efficiently, either. Instead of real-time computations, these operations, like square root units, can be offloaded to the CPU host and buffered in look-up tables, if the number of these operations is moderate compared to the problem domain size.

In the ISA-based PIM system, look-up tables are implemented with ordinary memory blocks, instead of customized hardware units. Contents of look-up tables will be loaded to the reserved memory blocks before the computation begins. The indexes for accessing look-up tables are generated in memory blocks during computation. Therefore, the procedure of look-up table access can be considered as a special case of inter-block data transmission, which can also be encoded as PIM instructions.

63 57	56 31	30 26	25 5	4 0
Opcode	Row ID	Offset_S	LUT Block ID	Offset_D

Figure 4: LUT Instruction Format

Algorithm 1 Steps for executing one look-up table instruction.

```
1: Generating read instruction $R\_1$ (location = Row Address × 1024 + Offset_S × 32, size = 32).
2: Issuing $R\_1$ to fetch the 32-bit *index*.
3: Generating read instruction $R\_2$ (location = LUT Block ID × 1024 × 1024 + *index* × 32, size = 32).
4: Issuing $R\_2$ to fetch the 32-bit content *data*.
5: Generating write instruction $W\_1$ (location = Row Address × 1024 + Offset_D × 32, size = 32, content = *data*).
6: Issuing $W\_1$.
```

Based on instruction formats used in prior work [33], we propose our look-up table instruction format (Figure 4). Bits 57-63 define the opcode, which differentiates look-up table instructions from other PIM instructions. The address that contains the look-up table index is calculated by shifting and adding the *Row ID* and $Offset\_S$ (assuming memory block size is 1024×1024, and the data precision is 32-bit, so only 5 bits are needed to define the offset). Afterwards, the address for the look-up table content will be calculated by adding the left shifted *LUT Block ID* and the obtained index. Finally, the content is transferred to the destination address, based on *Row ID* and $Offset\_D$. The detailed execution procedure of the look-up table instruction is listed in Algorithm 1.

5 WAVE SIMULATION IMPLEMENTATION

We introduce the dataflow of the acoustic and elastic wave simulation, and outline how it is mapped to our proposed Wave-PIM architecture for *Volume*, *Flux*, and *Integration* computations.

5.1 Data Layout and Execution Timeline

This part details the data layout in the PIM system and the execution timeline of *Volume* and *Flux*. *Integration* will not be discussed since it shares the same computation pattern as *Volume*.

We first provide an overview of the data layout of a 512-node element on a 1K×1K memory block (left part in Figure 5). We use the first 512 rows as computation spaces for each node in the element. The *variables*, *contributions*, and *auxiliaries* of each node are stored in the same columns. We use the other 512 rows as storage spaces for storing required constants of each element. These constants include *GLL Weight*, *GLL Point*, *dshape*, and *materials*. To make the computation of the 512 nodes processed in a row-parallel fashion, constants need to be copied to the scratchpad and broadcast to the first 512 rows before the computation begins.

Each memory block works independently. We use 4,096 memory blocks for the 4,096 elements of the refinement-level 4 acoustic wave simulation. The data layout of the elastic wave simulation needs to be adjusted. The 1K memory block row size is not enough for the nine variables in the elastic wave simulation, since enough scratchpads must be reserved for arithmetic operations [23] and the memory row size is limited due to circuit-level constrains [13]. As a result, we develop the expansion technique (Section 6.2) to use four memory blocks to deploy one element in the elastic model.

The execution timeline of *Volume* and *Flux* is shown in the right part of Figure 5. In *Volume*, *jacobian_det_w_star*, the grad p, and div $\mathbf{v}$ are first calculated by a series of addition and multiplication instructions after appropriate constants are distributed to each row. During the computation of grad p and div $\mathbf{v}$, the intermediate result *div_v*, which represents the divergence of the velocity vector, and *grad_p*, which represents the gradient of the pressure field, are generated and stored in the scratchpad. The *grad_p* is three-dimensional vector while *div_v* is scalar value.

Flux is implemented in a similar manner as *Volume*, while there are two major challenges. In *Flux*, inter-block data transmission is needed for obtaining *variables* from neighboring elements. The inter-block data movement is achieved by the H-tree or Bus interconnection, as discussed in Section 4.2. Another challenge in *Flux* computation is the presence of more complicated arithmetic operations, including square root and inverse, which exist when pre-processing the *materials*. We consider constant *materials* within an element. Since only two *materials* are used throughout each element, and each element has at most 6 neighboring elements, we offload the square root and inverse units to the CPU host. The pre-processed results are loaded to the look-up tables, as discussed in Section 4.3, before *Flux* computations begin. The alternative solution is to transfer the pre-processed data to each block directly. However, PIM memory blocks cannot read data from the off-chip memory during computation, making it unsuitable for pipelining as further optimization.

6 FITTING THE PROBLEM INTO THE PIM

We now introduce methods to adjust the data layout, in order to fit the problem into the available hardware resources. In some cases, PIM resources are insufficient, which necessitates processing of elements in batches, as we will discuss in Section 6.1. In addition, *variables* in one element can be expanded into multiple blocks in order to fully exploit the parallelism in acoustic wave simulation and also to make it possible to fit elastic wave simulation in our PIM systems, as outlined in Section 6.2. Furthermore, pipelining can be used to amortize the data fetching overhead when there is intensive inter-block data transmission.

6.1 Batching

In Section 5, we introduced how to implement each part of the acoustic wave simulation on the PIM system, with an assumption of unlimited PIM resources. In other words, there was no need to batch the elements due to insufficient memory capacity, which is not a realistic assumption. For example, if the refinement-level increases to 5, there will be 32,768 elements in the model, leading to a minimum of 4GB PIM capacity requirement. In this part, we discuss how to process large problems in batches.

6.1.1 ***Volume and Integration***. For *Volume* and *Integration*, batching simply means executing our initial solution multiple times, since there is no inter-element data dependency. As shown in Figure 6, we give an example of a two-batch procedure. Steps ①~④ are the original execution flows, which is the first batch in the folding implementation. For the second batch, step ①, i.e., broadcasting constants, can be removed, since the constants will not change during runtime. The overhead of batching is two additional transactions between off- and on-chip memory: store the outputs of the first batch and load the inputs of the second batch.

6.1.2 ***Flux***. For the *Flux*, the situation is much more complicated due to the data dependency between neighboring elements. When performing *Flux* computation on a pair of neighboring elements, both elements have to be stored on-chip. Take for example of deploying a refinement level 5 (containing $32 \times 32 \times 32$ elements) model on a 2GB PIM chip; only half of the elements can be stored on-chip. As shown in Figure 7, we assume that Slice 0 and 31 are the top and bottom slice, and Slice 0 ~15 are stored on-chip. In this situation, *Flux* computation for neighbors between Slice 15 and 16 cannot be achieved, since Slice 16 is not stored on-chip. Such edge cases have to be considered when batching the *Flux*.

With 3 axes, x, y, and z, and 2 normal vectors, -1 and $+1$, each element can have at most 6 neighbors, excluding the boundary elements. For the x axis and z axis, only intra-slice neighbors are considered and there is no inter-slice data transmission. Therefore, the computation of x and z axis in Slice 16 ~31 (Step ⑧, ⑨) can follow the same manner as Slice 0 ~15 (Step ②, ③), without modifying the execution flow. For the y axis, the data-flow has to be modified due to the inter-slice neighbors. With normal vector -1, we consider pairs of $(0, 1), (2, 3), ..., (30, 31)$ as neighbors. With normal vector +1, we consider pairs of $(1, 2), (3, 4), ..., (29, 30)$ as neighbors. For normal vector -1, the computation pattern does not need any change, since there is no interaction between Slice 15 and 16 (Step ④, ⑩). For normal vector +1, we first load the elements in Slice 16 to PIM (Step ⑤), then calculate Slice 1 ~16 (Step ⑥). In this way, *Flux* computation of a large model can be implemented on a small-scale PIM chip.

6.2 Expansion

In the batching technique, we assumed that we were using only one memory block for one element. In real cases, one element can

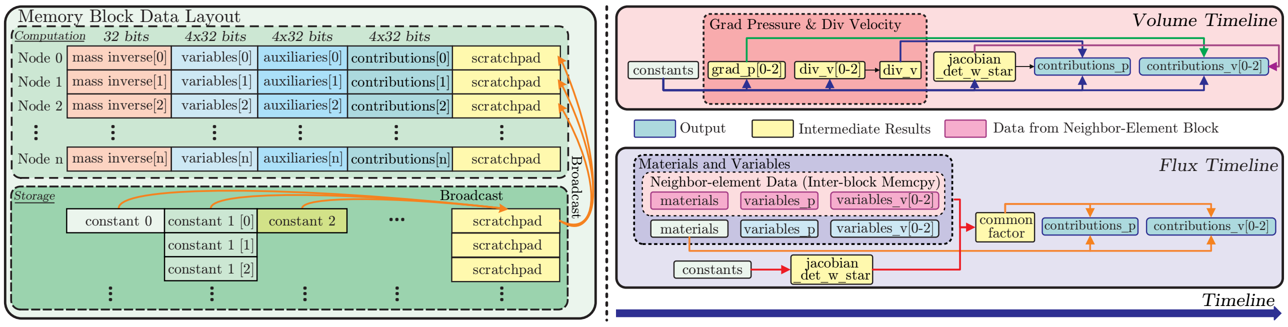

Figure 5: Single Element Data Layout and Execution Flow in PIM

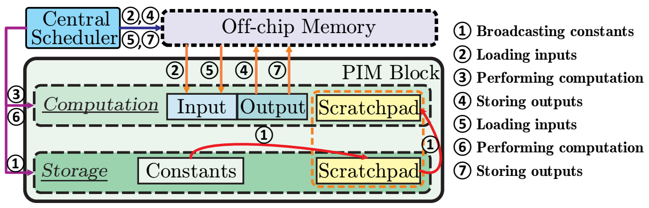

Figure 6: Batching of Volume and Integration

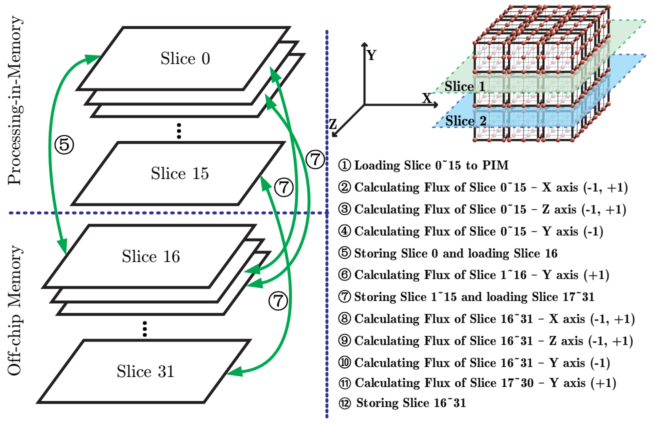
Processing-in-Memory
Off-chip Memory
Slice 0
Slice 15
Slice 16
Slice 31
Slice 1
Slice 2
Y
X
Z
① Loading Slice 0~15 to PIM
② Calculating Flux of Slice 0~15 – X axis (-1, +1)
③ Calculating Flux of Slice 0~15 – Z axis (-1, +1)
④ Calculating Flux of Slice 0~15 – Y axis (-1)
⑤ Storing Slice 0 and loading Slice 16
⑥ Calculating Flux of Slice 1~16 – Y axis (+1)
⑦ Storing Slice 1~15 and loading Slice 17~31
⑧ Calculating Flux of Slice 16~31 – X axis (-1, +1)
⑨ Calculating Flux of Slice 16~31 – Z axis (-1, +1)
⑩ Calculating Flux of Slice 16~31 – Y axis (-1)
⑪ Calculating Flux of Slice 17~30 – Y axis (+1)
⑫ Storing Slice 16~31

Figure 7: Batching of Flux

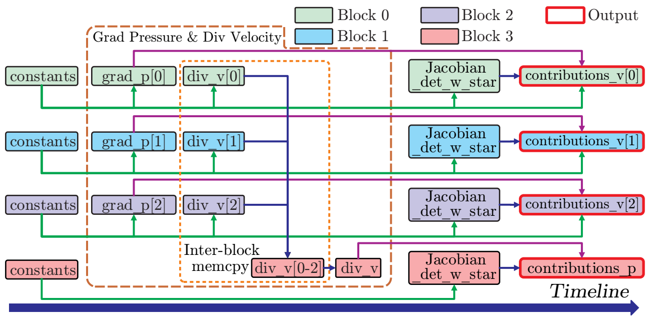

Figure 8: Four-block Implementation of Volume

also be deployed in multiple blocks. We name this technique as expansion, which is the opposite of batching, and we will discuss the expansion for both acoustic and elastic wave simulations.

6.2.1 ***Expansion in Acoustic Wave Simulation.*** In acoustic wave simulation, the computation of one element can be performed within one memory block. The four variables of each element are processed serially inside each block. However, some PIM resources may not be utilized when deploying a small refinement-level model on a large chip. For instance, deploying a refinement-level 4 model on a 2GB chip will only utilize 25% of available PIM resources, leading to low utilization of PIM resources that limits attainable performance while consuming more static power. To solve this, the computations of pressure p and velocity $\mathbf{v}$ can be distributed to four blocks (one for p and three for $\mathbf{v}$). These four variables will be processed in parallel instead, with an overhead of data duplication and inter-block data movement.

To expand the *Integration*, we simply distribute the four (one for p and three for $\mathbf{v}$) groups of *contributions* to four memory blocks, since there is no inter-block data dependency. The four groups of *auxiliaries* and *variables* will be generated in the respective block.

Expanding the *Volume* is much more complicated, since inter-block data dependency will exist if the computation of the grad p and div $\mathbf{v}$ is distributed into multiple blocks. Figure 5 shows the timeline of the one-block implementation, in which the computations of pressure and velocity, including *div_v*, *grad_p*, and *contributions*, are processed serially. For the expanded implementation (Figure 8), *jacobian_det_w_star* has to be calculated four times and constants have to be copied to the four blocks to provide the essential intermediate data. Additionally, *div_v* has to be transferred across blocks. With more dynamic power consumption, the four-block implementation can achieve a better performance than the one-block naive solution.

The four-block expansion implementation of *Flux* is shown in Figure 9. One block is used for buffering the required neighbor data *variables*. The reason is that, if the H-tree is applied, the inter-block data transmission latency within the four-block element will be shorter than long-distance inter-block transmission. The other three blocks are used for the computation of each axis. While buffering neighboring element data, part of the intermediate results can be computed at the same time. For example, the computation of *jacobian_det_w_star* does not require data from neighboring elements. Therefore, the overhead of buffering neighboring element data can be amortised.

6.2.2 ***Expansion in Elastic Wave Simulation.*** The elastic wave simulation is a little different from the acoustic one. Since nine

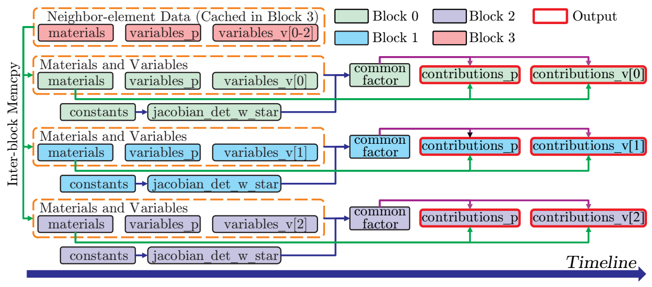

Figure 9: Four-block Implementation of Flux

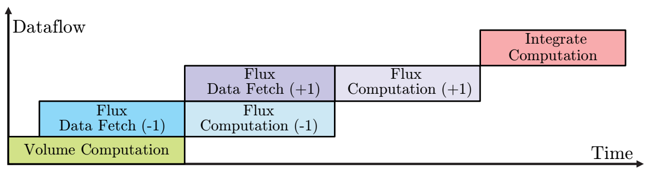

Figure 10: Dataflow Pipeline

variables of one element cannot be stored in one memory block, the expansion technique is required. The nine variables will be distributed to three or nine memory blocks. The *Integration* and *Flux* will follow the same manner as the expansion in the acoustic wave simulation. On the other hand, more inter-block `memcpy` in Figure 8 will happen for *Volume* in the elastic wave simulation using the same data transfer pattern.

6.3 Pipelining

The dataflow of the acoustic wave simulation can be further optimized with a pipelining technique. For the *Flux*, if additional PIM blocks are reserved for buffering the data from neighbor elements, pipelining can be done in 6 steps, since there are 3 axes and each axis has 2 normal vectors, and there is no data dependency between different axes and vectors. For the *Volume* and *Integration*, there is intra-block data transmission and data broadcasting. However, in PIM, both intra-block data movement and computation are implemented by applying different voltages on bitlines and wordlines. This hardware hazard makes the *Volume* and *Integration* unable to be pipelined. In addition to the intra-kernel pipeline in *Flux*, pipelines can be used for inter-kernel if there is no data dependency. For example, the neighboring-element data fetching in *Flux* and the computation in *Volume* can be processed in parallel (Figure 10).

7 EVALUATION

We now evaluate the efficacy of our proposed Wave-PIM solution for wave simulations. We implement six benchmarks on three real GPU platforms, and one cycle-accurate PIM simulator, to evaluate the performance improvement and energy savings of our PIM systems in comparison to GPUs. Batching and expansion techniques are included in the comparisons to show the scalability of our proposed solutions. As addition, the detailed performance improvement by pipelining will be discussed by a breakdown analysis. Trade-offs between the H-tree and Bus interconnection will also be presented.

7.1 Experiment Setup

We create a simulation framework for Wave-PIM by adapting NVSim [13] and FloatPIM [26] . We choose a 32-bit floating point data precision for both PIM and GPU baseline. Moreover, complicated arithmetic operations, such as square root and inverse operations, are offloaded to the host CPU for the case of PIM.

Various PIM basic operation parameters are referenced from FloatPIM [26], which are listed in Table 4. The configuration of the 2GB PIM chip listed in Table 3, which is referred from DUAL [27]. We keep the Crossbar Array size as 1k*1K for the other three PIM settings (512MB, 8GB, 16GB) and only increase/decrease the number of tiles. To support our proposed ISA, we add an additional decoder logic inside each block, and one central controller. One host CPU (we assume an ARM Cortex-A72 architecture [19]) has to be used for sending instructions and pre-processing part of the input data. We assume a 900GB/s HBM2 DRAM as the off-chip memory for our Wave-PIM, where the power of the off-chip memory is 36.91W [34]. To support the inter-block data transmission, we add H-tree/Bus switches inside each memory tile for data routing. We use the Synopsys PrimeTime [3] tool to obtain the power consumption of other circuits, and list them in Table 3.

We use three state-of-the-art GPU platforms (Nvidia GTX 1080Ti, Tesla P100, and Tesla V100) as the baseline to compare with our proposed PIM solution. The hardware configuration details are shown in Table 2. The maximum throughput for the GPU platforms is obtained from Nvidia whitepapers [9, 10]. The throughput for our PIM system is calculated based on the maximum parallelism

Table 2: Hardware Configurations

Platform	GPU	GPU	GPU	PIM
Name	GTX 1080Ti	Tesla P100	Tesla V100	Wave-PIM
Host CPU Model	Xeon E5-2620 v4	Xeon Platinum 8160	Xeon Platinum 8160	ARM Cortex-A72
Process Node	16nm	16nm	12nm	28nm
Clock Frequency	1,530MHz	1,480MHz	1,582MHz	900MHz
Register Size	7,168KB	14,336KB	20,480KB	N/A
L2 Cache Size	2,816KB	4,096KB	6,144KB	N/A
Memory Size and Type	11GB GDDR5X	16GB HBM2	16GB HBM2	512MB, 2GB, 8GB, 16GB
Memory BW	484GBps	720GBps	900GBps	900GBps
FP32 CUDA Cores	3,584	3,584	5,120	N/A
FP32 Peak Throughput	11.3TFLOPS	10.6TFLOPS	15.7TFLOPS	1.7, 6.6, 26.4, 52.8 TFLOPS

Table 3: PIM Parameters (2GB capacity)

Components	Param	Value	Power
Crossbar Array	size	1Mb	6.14mW
Sense Amp	number	1K	2.38mW
Decoder	number	1	0.31mW
Memory Block	number	1	8.83mW
Tile Memory	num_block	256	1.57W
H-tree Switch	number	85	107.13mW
Bus Switch	number	1	17.2mW
Tile	Size	32MB	1.68W (H-tree) 1.59W (Bus)
Central Controller	number	1	6.41W
CPU Host	number	1	3.06W
Total	Size	2GB	115.02W (H-tree) 109.25W (Bus)

Table 4: PIM Basic Operation Energy (E) and Time (T)

E_{set}	E_{reset}	E_{NOR}	E_{search}	T_{NOR}	T_{search}
23.8fJ	0.32fJ	0.29fJ	5.34pJ	1.1ns	1.5ns

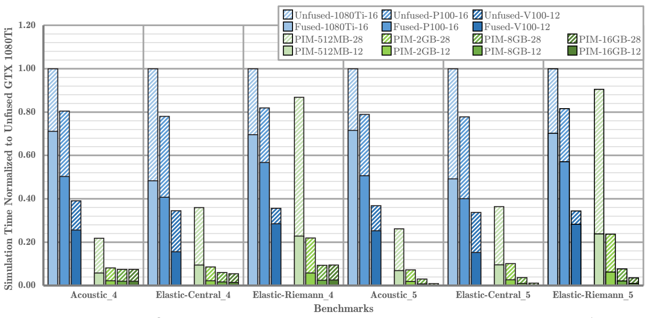

Figure 11: Performance Comparison Between GPU and PIM

($2GB/1,024b = 16M$) and the arithmetic operation latency from prior works [23, 26], assuming a workload containing 50% addition and 50% multiplication operations. We use the Nvidia-SMI [8] tool and Intel's RAPL [11] tool to collect the energy consumption on the GPU platforms and their hosts.

7.2 Benchmarks

To evaluate performance and energy consumption of both GPU and PIM implementations, we use six benchmarks divided into three groups: acoustic wave simulation, elastic wave simulation with central flux solver, and elastic wave simulation with Riemann flux solver. Each group has two different refinement-levels which define the number of elements needed to simulate. The Table 6 describes the main features of these six benchmarks. The performance is measured based on the simulation time while the energy is measured from the total power consumption of both host CPU and accelerator (i.e., GPU or PIM).

The benchmarks have total number of instructions in the range of 2 billion to 79 billion and total number of single-precision floating point operations in the range of 400 million to 12 billion. These numbers are from fused GPU implementation obtained using `nvprof` [7] where each kernel launched once on Tesla V100 GPU. Usually, the simulation run for thousands of time-steps where each time-step requires each kernel to be launched five times.

The unfused GPU implementation runs on GTX 1080Ti is used as the baseline of our evaluation for both performance and energy consumption. This implementation consists of three primary kernels that compute *Volume*, *Flux*, and *Integration* as shown in Figure 2. Some optimizations include storing repeatedly-used values in constant-memory, rearranging code to minimize divergence, and extracting element-level and node-level parallelism.

The fused GPU implementation is one step further to optimize the GPU implementation that uses kernel fusion to achieve higher performance. This implementation merges compute *Volume* and compute *Flux* into single kernel to minimize the data movements. This version also incorporates more efficient algorithms to determine the neighboring elements and more data locality for each thread since it handles one node throughout kernel execution.

Table 5: PIM Implementation Configuration

Configuration	512MB	2GB	8GB	16GB
Acoustic_4	N	E_a	E_a	E_a
Elastic_4	E_e&B	E_e	E_a&E_e	E_a&E_e
Acoustic_5	B	B	N	E_a
Elastic_5	E_e&B	E_e&B	E_e&B	E_e

N represents the naive implementation, E_a represents implementation using expansion to increase the parallelism (can be used for both acoustic and elastic wave simulation), E_e represents implementation using expansion due to limited row size (only exists in the elastic wave simulation) , and B represents implementation using the batching technique.

Table 6: Characteristic of Benchmarks Used for Evaluation

Benchmark	Refinement Level	Number of Elements	Number of Instructions[1]	Number of FP Ops.[2]
Acoustic_4	4	4,096	2,140,930,048	391,380,992
Elastic-Central_4	4	4,096	3,465,543,680	990,117,888
Elastic-Riemann_4	4	4,096	9,870,131,200	1,472,200,704
Acoustic_5	5	32,768	17,127,440,384	3,131,047,936
Elastic-Central_5	5	32,768	27,724,349,440	7,920,943,104
Elastic-Riemann_5	5	32,768	78,960,159,424	11,777,661,440

[1] From `inst_executed` metric multiplied by 32 to obtain thread-level value.
[2] From `flop_count_sp` and `flop_count_sp_special` metrics.
[1,2] Obtained using `nvprof` running on Tesla V100 with fused implementation. Values are the total from each kernel launched once. In each time-step, each kernel is launched five times.

7.3 Performance Comparisons

In this section, we compare the performance between the Wave-PIM and GPU baseline. Our PIM simulations are done with process node of 28nm, whereas the GPU numbers are obtained from the real hardware fabricated on 16nm or 12nm process node. Approximate scaling as suggested by [2, 50] indicate that with 12nm process node, the PIM can get 3.81× performance improvement and 2.0× energy savings. We report both unscaled and scaled performance results in Figure 11. The batching and expansion techniques for the PIM implementation are also considered, which are listed in Table 5.

Due to the reduction in data movement, our Wave-PIM can achieve up to 414.37× speedup compared with the GPU implementation. On average, our four Wave-PIM solutions achieve 10.28×, 35.80×, 72.21×, and 172.76× speedups on the six benchmarks compared with the Unfused-1080Ti baseline. Compared against the most efficient GPU implementation (Fused-V100), our four Wave-PIM solutions can still achieve 2.30×, 7.89×, 15.97×, and 37.39× throughput. The speedup of Elastic-Riemann on PIM is below the average because the computation intensity of Elastic-Riemann is high, and thus the performance improvement on PIM due to the reduction of data movement is insignificant compared to the Acoustic and Elastic-Central benchmarks. The 512MB PIM solution does not perform well since the inputs have to be divided into 32 batches for the refinement-level 5 of elastic wave simulation, which results in additional data movement from DRAMs.

7.4 Energy Comparisons

We report the energy comparisons between our PIM solutions and GPU baseline in Figure 11. If the PIM is large enough to store the size of problem (e.g. 512MB PIM for the refinement-level 4 of acoustic wave simulation), our Wave-PIM can achieve at most 50.56× energy savings compared to the GPU implementation. On average, our four Wave-PIM solutions achieve 26.62×, 26.82×, 14.28×, and 16.01× energy savings on the six benchmarks compared to the Unfused-1080Ti baseline.

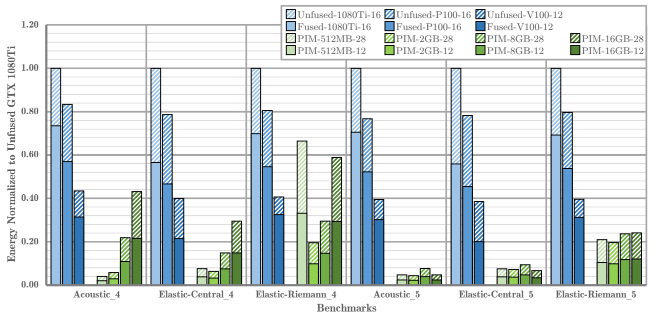

Figure 12: Energy Comparison Between GPU and PIM

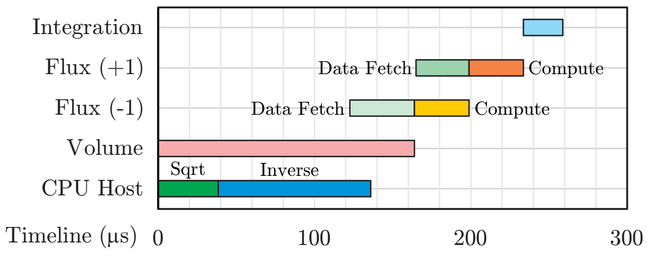

Figure 13: Pipeline Breakdown

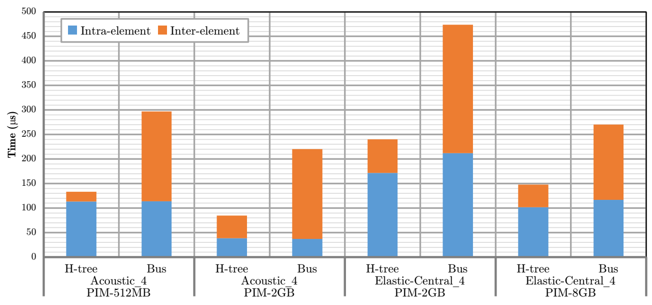

Figure 14: Comparison between H-Tree and Bus

There is a trade-off between the performance and energy consumption. With a large PIM chip, large problems can benefit from the abundant parallelism and zero overhead DRAM data transfer since batching is not needed. However, small problems may not be able to take performance advantage of large PIM chip because of the resources under-utilization, which is the reason for less energy savings compared to small chips.

7.5 Pipeline Analysis

We now discuss performance improvement if pipelining is applied in the Wave-PIM system (Figure 13). The square root and inverse operations need to be pre-processed for the *Flux* by offloading them to the host CPU during the *Volume* computation. At the same time, the neighboring element data can be transferred to Block 3 (Figure 9) before the computation of *Flux* begins. We divide the computation in *Flux* based on the direction of normal vector into two stages in order to overlap the overhead of inter-block data transmission. Without pipelining, our Wave-PIM can only obtain a 0.77× throughput (Figure 11).

7.6 Comparison between Bus and H-tree

The performance of the Bus interconnection is the same as the H-tree interconnection, if there is a few inter-block data transmission, as in the *Volume* and *Integration*. The Bus interconnect can benefit from the lower power compared to the H-tree. However, if there is an intensive inter-block interactions, the performance difference is more prominent. In this part, we will provide case studies for the *Flux*, since a huge amount of inter-block data transmission is required between neighboring elements.

Figure 14 shows the performance comparison between H-tree and Bus in four cases. Without expansion (Acoustic-512MB and Elastic-2GB), the inter-element data transmission contributes 21.62% 58.41% to the overall execution time for the H-tree and Bus, respectively. Moreover, if expansion is applied (Acoustic-2GB and Elastic-8GB), the two ratios will increase to 42.77% and 69.96%. By considering the pipelines in Figure 13, inter-element data transmission time should be shorter than the intra-element computation time, and thus we used the H-tree interconnection in our design.

8 CONCLUSION

Processing-in-Memory reduces data movement, and allows harnessing the data-level parallelism that is abundant in many scientific computing applications. In this paper, we mapped the six wave simulation benchmarks on a PIM system. To maximize the benefits of processing in memory, the dependencies and movement of the data and intermediate results between computation steps were carefully analyzed and handled. We also introduced the concept of batching and expansion, in order to improve the scalability of the Wave-PIM implementation to support problems of various sizes at the highest possible performance. In addition, we also proposed and evaluated H-tree interconnection against the Bus interconnection to facilitate more efficient inter-block data transfer. All of these combined yielded not only performance gains, but also reduction in data movement, which decreases the energy consumption. Experimental results show that our PIM solution significantly outperforms state-of-the-art GPU systems, yielding an average of 41.98× speedup, and 12.66× energy savings.

ACKNOWLEDGMENTS

This research was supported in part by ExxonMobil Research and Engineering Company, agreement no. EM10480.36, National Science Foundation (NSF) grant number 1763848, and computational resources from Texas Advanced Computing Center (TACC). Any opinions, findings, conclusions or recommendations are those of the authors and not of the sponsors.

REFERENCES

[1] Aria Abubakar, Gong Li Wang, Lin Liang, Tarek M. Habashy, and Maokun Li. 2016. Electromagnetic modeling and inversion application for oil and gas industry. In *2016 Progress in Electromagnetic Research Symposium (PIERS)*. 938–938. https://doi.org/10.1109/PIERS.2016.7734532
[2] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, et al. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 715–731.
[3] Himanshu Bhatnagar. 2002. *Advanced ASIC chip synthesis*. Springer.

[4] Jesse Chan, Zheng Wang, Axel Modave, Jean-Francois Remacle, and T. Warburton. 2016. GPU-accelerated discontinuous Galerkin methods on hybrid meshes. *J. Comput. Phys.* 318 (2016), 142 – 168.

[5] M. V. K. Chari. 1983. Electromagnetic Modeling of Electrical Machinery for Design Applications. In *Industrial Electromagnetics Modelling*, J. Caldwell and R. Bradley (Eds.). Springer Netherlands, Dordrecht, 3–14.

[6] J. Chen and Q. H. Liu. 2013. Discontinuous Galerkin Time-Domain Methods for Multiscale Electromagnetic Simulations: A Review. *Proc. IEEE* 101, 2 (2013), 242–254.

[7] Nvidia Corporation. 2019. Nvidia CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html/.

[8] Nvidia Corporation. 2020. Nvidia System Management Interface. https://https://developer.nvidia.com/nvidia-system-management-interface/.

[9] Nvidia Corporation. 2020. Nvidia Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[10] Nvidia Corporation. 2020. Nvidia Turning GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[11] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE, 189–194.

[12] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1–8.

[13] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. 2012. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (2012), 994–1007.

[14] T. Duda, J. Bonnel, E. Coelho, and K. Heaney. 2019. Computational Acoustics in Oceanography: The Research Roles of Sound Field Simulations. *Acoustics Today* 15 (2019), 28–37. Issue 3.

[15] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 383–396.

[16] Arash Fathi, Loukas F Kallivokas, and Babak Poursartip. 2015. Full-waveform inversion in three-dimensional PML-truncated elastic media. *Computer Methods in Applied Mechanics and Engineering* 296 (2015), 39–72.

[17] Arash Fathi, Babak Poursartip, and Loukas F. Kallivokas. 2015. Time-domain hybrid formulations for wave simulations in three-dimensional PML-truncated heterogeneous media. *Internat. J. Numer. Methods Engrg.* 101, 3 (2015), 165–198.

[18] Arash Fathi, Babak Poursartip, Kenneth H. Stokoe II, and Loukas F. Kallivokas. 2016. Three-dimensional P- and S-wave velocity profiling of geotechnical sites using full-waveform inversion driven by field data. *Soil Dynamics and Earthquake Engineering* 87 (2016), 63 – 81.

[19] Raspberry Pi Foundation. 2021. Raspberry Pi Power Supply Documentation. https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md/.

[20] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 100–113.

[21] DC Groeneveld, LKH Leung, PL Kirillov, VP Bobkov, IP Smogalev, VN Vinogradov, XC Huang, and E Royer. 1996. The 1995 look-up table for critical heat flux in tubes. *Nuclear Engineering and design* 163, 1-2 (1996), 1–23.

[22] Lluís Guasch, Oscar Calderon Agudo, Meng-Xing Tang, Parashkev Nachev, and Michael Warner. 2020. Full-waveform inversion imaging of the human brain. *npj Digital Medicine* 3 (2020), 1 – 12.

[23] Ameer Haj-Ali, Rotem Ben-Hur, Nimrod Wald, and Shahar Kvatinsky. 2018. Efficient algorithms for in-memory fixed point multiplication using magic. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[24] J.S. Hesthaven and T. Warburton. 2010. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer. http://books.google.com/books?id=RQWvcQAACAAJ

[25] S. B. Hong, N. Vlahopoulos, R. M. Mantey, and D. J. Gorsich. 2004. A computational approach for evaluating the probability of acoustic detection of a military vehicle. In *Targets and Backgrounds X: Characterization and Representation*, Wendell R. Watkins, Dieter Clement, and William R. Reynolds (Eds.), Vol. 5431. International Society for Optics and Photonics, SPIE, 150 – 159.

[26] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 802–815.

[27] Mohsen Imani, Saikishan Pampana, Saransh Gupta, Minxuan Zhou, Yeseong Kim, and Tajana Rosing. 2020. DUAL: Acceleration of Clustering Algorithms using Digital-based Processing In-Memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 356–371.

[28] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2015. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing* 37, 5 (2015), C497–C531. https://doi.org/10.1137/140970963

[29] L.F. Kallivokas, A. Fathi, S. Kucukcoban, K.H. Stokoe, J. Bielak, and O. Ghattas. 2013. Site characterization using full waveform inversion. *Soil Dynamics and Earthquake Engineering* 47 (2013), 62 – 82.

[30] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (2011), 7–17. https://doi.org/10.1109/MM.2011.89

[31] Martin-D Lacasse, Laurent White, Huseyin Denli, and Lingyun Qiu. 2018. Full-wavefield inversion: An extreme-scale PDE-constrained optimization problem. In *Frontiers in PDE-Constrained Optimization*. Springer, 205–255.

[32] Charles E Leiserson. 1980. Area-efficient graph layouts. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE, 270–281.

[33] Ruihao Li, Shuang Song, Qinzhe Wu, and Lizy. K John. 2020. Accelerating Force-directed Graph Layout with Processing-in-Memory Architecture. In *2020 27th IEEE International Conference on High Performance Computing, Data, & Analytics (HiPC)*. IEEE.

[34] Shang Li, Dhiraj Reddy, and Bruce Jacob. 2018. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*. 341–353.

[35] Elena Lucano, Micaela Liberti, Gonzalo G. Mendoza, Tom Lloyd, Maria Ida Iacono, Francesca Apollonio, Steve Wedan, Wolfgang Kainz, and Leonardo M. Angelone. 2016. Assessing the Electromagnetic Fields Generated By a Radiofrequency MRI Body Coil at 64 MHz: Defeaturing Versus Accuracy. *IEEE Transactions on Bio-Medical Engineering* 63, 8 (8 2016).

[36] Mark E Lucente. 1993. Interactive computation of holograms using a look-up table. *Journal of Electronic Imaging* 2, 1 (1993), 28–35.

[37] Babak Poursartip, Arash Fathi, and Loukas F. Kallivokas. 2017. Seismic wave amplification by topographic features: A parametric study. *Soil Dynamics and Earthquake Engineering* 92 (2017), 503–527. https://doi.org/10.1016/j.soildyn.2016.10.031

[38] Babak Poursartip, Arash Fathi, and John L. Tassoulas. 2020. Large-scale simulation of seismic wave motion: A review. *Soil Dynamics and Earthquake Engineering* 129 (2020), 105909.

[39] A. Quarteroni and A. Valli. 1994. *Numerical Approximation of Partial Differential Equations*. Springer.

[40] A. Quarteroni and A. Valli. 2008. *Numerical approximation of partial differential equations*. Springer, Berlin Heidelberg.

[41] K. Sankaran. 2019. Recent trends in computational electromagnetics for defence applications. *Defence Sience Journal* 69 (2019), 65–73. Issue 1.

[42] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[43] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 541–552.

[44] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology* 15, 4 (2016), 635–650.

[45] G.A.E. Vandenbosch. 2004. Computational electromagnetics and antenna design in Western Europe-organisation and overview of the present status. In *The Fifth International Kharkov Symposium on Physics and Engineering of Microwaves, Millimeter, and Submillimeter Waves (IEEE Cat. No.04EX828)*, Vol. 1. 40–45 Vol.1. https://doi.org/10.1109/MSMW.2004.1345785

[46] L.C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas. 2010. A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media. *J. Comput. Phys.* 229, 24 (2010), 9373 – 9396.

[47] 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.

[48] Xingyao Zhang, Shuaiwen Leon Song, Chenhao Xie, Jing Wang, Weigong Zhang, and Xin Fu. 2020. Enabling highly efficient capsule networks processing through a PIM-based architecture design. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 542–555.

[49] Yuhao Zhang, Zhiping Jia, Yungang Pan, Hongchao Du, Zhaoyan Shen, Mengying Zhao, and Zili Shao. 2020. PattPIM: A Practical ReRAM-Based DNN Accelerator by Reusing Weight Pattern Repetitions. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[50] Scott Zuloaga, Rui Liu, Pai-Yu Chen, and Shimeng Yu. 2015. Scaling 2-layer RRAM cross-point array towards 10 nm node: A device-circuit co-design. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 193–196.