Accelerating Force-directed Graph Layout with Processing-in-Memory Architecture

Ruihao Li*, Shuang Song†, Qinzhe Wu*, Lizy K. John*
*The University of Texas at Austin †Facebook Inc.
liruihao@utexas.edu, shuangsong@fb.com, qw2699@utexas.edu, ljohn@ece.utexas.edu

***Abstract*—In the big data domain, the visualization of graph systems provides users more intuitive experiences, especially in the field of social networks, transportation systems, and even medical and biological domains. Processing-in-Memory (PIM) has been a popular choice for deploying emerging applications as a result of its high parallelism and low energy consumption. Furthermore, memory cells of PIM platforms can serve as both compute units and storage units, making PIM solutions able to efficiently support visualizing graphs at different scales. In this paper, we focus on using the PIM platform to accelerate the Force-directed Graph Layout (FdGL) algorithm, which is one of the most fundamental algorithms in the field of visualization. We fully explore the parallelism inside the FdGL algorithm and integrate an algorithm level optimization strategy into our PIM system. In addition, we use programmable instruction sets to achieve more flexibility in our PIM system. Our PIM architecture can achieve $8.07\times$ speedup compared with a GPU platform of the same peak throughput. Compared with state-of-the-art CPU and GPU platforms, our PIM system can achieve an average of $13.33\times$ and $2.14\times$ performance speedup with $74.51\times$ and $14.30\times$ energy consumption reduction on six real world graphs.**

I. Introduction

The big data era is changing and shaping all domains of modern life including healthcare, retail and finance. Graph systems are widely used to process big data [34], [24], [38], [33]. To give a more intuitive illustration of graph systems and allow users to better understand the data contained inside them, visualization techniques are developed to draw graphs in a 2D or 3D space. Graph visualization techniques have been widely applied in different fields, such as transportation systems [10], biology [11], and energy [31]. Fig. 1 gives examples of visualizing road map systems and medical systems in graphs. A more efficient hardware solution with high performance and low energy consumption for graph visualization systems is more desired nowadays, since navigation systems are usually deployed in edge devices with computing power constraints, and the latency is a key factor in medical visualization systems for doctors making judgements during surgeries.

In the graph visualization system, each vertex of the graph is allocated with specific coordinates, and connected with other vertices by straight or curved lines. Force-directed Graph Layout (FdGL), which we will focus on in this paper, is one of the most fundamental graph layout algorithms. It uses solid points to depict vertices and uses straight lines to depict connections between vertices [25], [6], [35], [36]. A large proportion of advanced visualization methods utilize the same computation pattern as the FdGL and use curved edges or group nearby vertices to show the graph in a more balanced state [20], [18], [14], making it necessary to find a more efficient way to accelerate FdGL.

(a) Road map system. [18]

(b) Biology system. [11]

Fig. 1: Examples of visualization systems.

Due to high similarity between these advanced visualization algorithms and the FdGL, they face the same performance challenges. In visualization systems, to obtain a more intuitive layout, the coordinates of each vertex have to be updated frequently. As a result, the generated intermediate results, the size of which scale with the size of the input graph, require a huge amount of on-chip memory to store them considering the scales of real world graphs (thousands and millions of vertices and edges) [44], [17]. Data movement between on-chip and off-chip memory is unavoidable due to the limited on-chip memory capacity. Because of the large overhead of data movement, accelerating these visualization applications remain great challenges.

Prior works have proposed optimizations to accelerate FdGL from different perspectives, including using simulated annealing to speed up the convergence of the algorithm [30] or using tree based data structures to prune less useful computations [36]. Since the coordinates of each vertex have to be updated after each iteration until reaching equilibrium, applying algorithm-level optimization focused on accelerating convergence or reducing computation scale requires significant control logic for data synchronization. As a result, hardware accelerators which do not support irregular control flow may not be able to apply these algorithm level optimizations efficiently. In order to avoid the overhead of data synchronization, prior GPU solutions only tried to adjust the dataflow and computation sequence of FdGL to explore a higher degree of parallelism [8], [39], but algorithm-level optimizations were not considered. These two optimizations only focused on decreasing the time complexity or exploring the parallelism of FdGL, without paying so much attention to the tremendous amount of intermediate data movement between on-chip and off-chip memories. To solve these issues, we aim to integrate algorithm-level optimizations into the design of a hardware accelerator with high parallelism and low data movement.

In this paper, we propose a customized Processing-in-Memory (PIM) system to speedup the algorithm-level optimized FdGL with high parallelism and reduced data move-

ment. In our proposed PIM system, basic computational operations are abstracted as instructions and are decoded by control logic inside the memory chip. This makes PIM systems more flexible than GPUs and enables tree based data structures implemented inside memory for computation pruning. We achieve these advanced data structures with our software defined instruction sets instead of specialized hardware units. This makes our design suitable for graphs of different sizes without loss of generality compared to GPU platforms. We summarize the contributions of this paper as follows:

- We present a PIM design for the FdGL with minimal additional hardware, and create a system that supports datasets of different sizes and the quadtree/octree data structure. Instruction sets are also included in the system.
- To our best knowledge, we are the first to deploy division and trigonometric units in digital PIM. These units are not fully explored in previous digital PIM researchs.
- We design an analytical model to analyze the performance of CPU, GPU and PIM architectures in a uniform manner. We also introduce an iso-throughput GPU platform providing the same maximum throughput as PIM to reduce the architectural difference between the platforms in comparison. Based on the calculated results, our PIM architecture achieves a speedup $8.07\times$ than the iso-throughput GPU platform.
- We evaluate our PIM design with extensive experiments on six real graphs of various sizes. Compared to state-of-the-art CPU and GPU platforms, our PIM system yields a performance increase $13.33\times$ and $2.14\times$ over a state-of-the-art CPU (Xeon Platinum) and GPU (1080 Ti) respectively, and energy savings of $74.51\times$ and $14.30\times$ over these platforms after applying software level optimizations.

II. Background and Motivation

This section introduces the FdGL algorithm and popular acceleration strategies. The performance bottlenecks of current hardware solutions to accelerate FdGL will be discussed to motivate our proposed design. Finally, we will also talk about basics about PIM and illustrate why it is a better candidate than other hardware platforms.

A. Introduction to Force-directed Graph Layout

The FdGL is one of the basic algorithms for visualizing graphs in 2D or 3D spaces [6], [35], [36]. It evolved from a VLSI technique called Force-directed placement [25]. It uses points to represent vertices in graphs and straight lines to represent edges. There are two kinds of forces in graphs, which are repulsive forces and spring forces; the coordinates of all vertices are adjusted iteratively to achieve equilibrium. The repulsive force exists between each pair of vertices in the graph, while the spring force only exists between each pair of connected vertices.

The repulsive force is derived from Coulomb's Law, it can be written as:

$$F_r = C_3/d^2 \tag{1}$$

where C_3 is a constant and d represents the distance between two vertices. Secondly, the spring force was inspired by Hooke's Law, which is written as:

$$F_s = C_1 * log(d/C_2) \tag{2}$$

The aforementioned forces are useful in building graph visualization models. Each node is initially assigned with random coordinates. The system is usually not stable at first, meaning that most of the vertices are not in a state of stress equilibrium. We need to move the vertices step by step until reaching a state of relative stress equilibrium.

B. Software Optimization

The time complexity of calculating the repulsive force and spring force inside each iteration are $O(|V|^2)$ and $O(|E|)$ separately, where $|V|$ represents the number of vertices and $|E|$ represents the number of edges. Most real graph and 3D model datasets are sparse [44], making the repulsive force the bottleneck of the whole system. To address this issue, prior work tried to use the grid-variant algorithm [36], dividing the screen into a grid of squares. Vertices in nearby squares of the grid instead of the whole graph are considered when computing the repulsive force, since distant vertices contribute little to the resultant repulsive force. Moreover, others tried to use tree based data structures [45] to implement the grid-variant algorithm. With tree based acceleration strategies, the time complexity of repulsive force calculation can be decreased to $O(|V|log|V|)$. The overhead of introducing the tree based data structure is inserting each tree node to the quadtree (in 2D space) or octree (in 3D space) at the beginning of each iteration. The time complexity of inserting the $|V|$ vertices into the quadtree is $O(|V|log|V|)$, which is the same as calculating the repulsive force. In this way, the time complexity of each iteration can be decreased from $O(|V|^2)$ to $O(|V|log|V|)$ for current graph datasets considering their sparsity.

C. Challenge of Current GPU Implementation

In this part, we will discuss the inefficiencies of utilizing state-of-the-art GPU platforms implementing the FdGL, especially when tree based optimization strategies are applied. To give a quantitative analysis, we implemented FdGL of the graph [12] on Nvidia Tesla V100 GPU platform (results in Fig. 2). We used nvprof [2] to get the kernel execution time for better illustrate the potential bottleneck of the GPU implementation.

When quadtree/octree based optimization is applied, the computation of the repulsive force in each iteration cannot start before reconstructing the quadtree/octree. Different from computation of the forces, the update and reconstruction of quadtree/octree contains less parallelism. Additionally, tree nodes in the quadtree/octree have to be splitted into child nodes when they reach the maximum tree node capacity, so an atomic variable is needed for each tree node to store the current size. Due to these synchronization events, the intermediate results generated after each iteration have to be transferred from the GPU to the CPU host for updating the

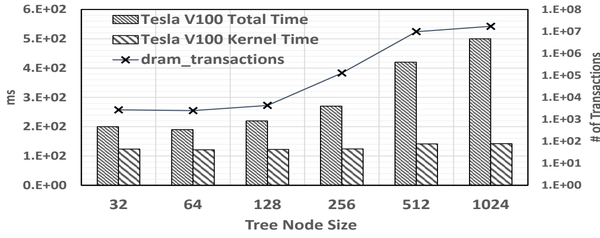


Fig. 2: Limitations of FdGL GPU implementation.

quadtree/octree, and then sent back to the GPU for the next iteration.

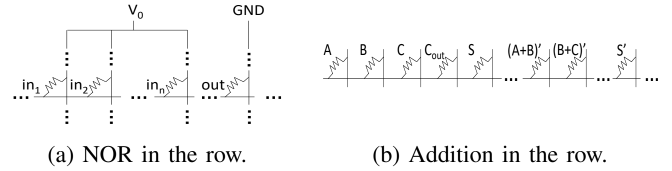
As shown in Fig. 2, the overhead of data movement was as large as 71.57% of the overall GPU processing time, which is 17,047K DRAM transactions (corresponding to tree node size 1024). We notice that the overhead can be reduced as the tree node size (the maximum number of vertices that can be contained in one rectangular grid of the 2D space) decreases. However, we cannot make the tree node size as small as we desire. Each vertex is only affected by the repulsive force from the vertices in the same tree node. As a result, smaller tree node size means more vertices (which are relatively further away from that vertex) will be neglected for the repulsive force computation and will result in higher accuracy loss. To accommodate these optimized tree based algorithms, a solution with a high degree of parallelism but little data movement is needed.

D. Other Possible Solutions

Although ASICs and FPGAs could appear to be good candidates for deploying visualization systems because of the possibility of maintaining the tree based data structure on chip, there are other challenges.

The performance of ASIC platforms fluctuates between large and small datasets. For the computation of repulsive forces, the time complexity and scale of intermediate results are highly dependent on the scale of graphs we choose. For large datasets, to reduce the overhead of data movement, large on-chip buffers for storing intermediate results are required. Since each hardware unit has to be fixed in an ASIC platform and cannot be reconfigured, small datasets might lead to severe under-utilization of resources. This limits the average performance of ASIC platforms.

FPGAs might perform well for graphs of diverse sizes because of abundant on-chip logic resources, but visualization systems are highly memory bounded applications. The coordinates of each vertex have to be updated after each iteration, and intermediate results are generated during the computation of forces. The amount of on-chip memory currently in FPGA platforms (Mb level) [46] is inadequate to handle large data sets [44] without moving large amounts of data between on-chip and off-chip memory. This makes FPGA platforms not a suitable candidate for FdGL. A solution with large on-chip memory capacity as well as abundant and flexible control logic is desired.



(a) NOR in the row. (b) Addition in the row.

Fig. 3: PIM operations in rows.

E. Processing-in-memory Basics

PIM is becoming an alternative candidate for emerging big data applications by reducing data movement between processing units and memory units [9], [23]. In digital PIM systems, the resistance of each memristor cell can switch between R_{ON} and R_{OFF}, which represents '1' and '0' in logic separately, when different voltages are applied to the bitline or wordline. This property can be exploited to implement a NOR gate in the digital memory [26]. As shown in Fig. 3a, the output memristor is initialized to R_{ON}. If one or more of the n inputs switched from '0' to '1', the output memristor will switch from R_{ON} to R_{OFF}. This operation performs the same functionality as a NOR gate, as shown is Fig. 3a. With basic NOR units, arithmetic units including addition [7] can be achieved by performing NOR operations sequentially. The arithmetic addition can be expressed as:

$$C_{out} = ((A+B)' + (B+C)' + (C+A)')' \quad (3a)$$

$$S = (((A'+B'+C')' + ((A+B+C)' + C_{out})')')' \quad (3b)$$

Where A, B represent the two input bits, C represents carry in bit, C_{out} represents carry our bit, and S represents sum bit. The logic for the addition operation is shown in Fig. 3b. Furthermore, arithmetic multiplication units can also be achieved in digital PIM systems in a similar way by performing additions sequentially [7].

F. Advanced PIM Architecture

With the aforementioned arithmetic units, current PIM systems can be applied to deploy many kinds of applications including machine learning [9], [23], [40], graph processing [41], and even block chain [29]. However, there is a lot more to be desired for PIM systems.

Prior PIM works highly relied on customized control logic for specific parts of the application. Under this circumstance, it is difficult for these prior PIM designs to find an one-fit-all solution achieving the theoretical maximum performance. For example, in machine learning systems, based on the roof-line model, some network layers are memory intensive while others are computing intensive [15]. Without dynamic configuration of hardware units, PIM systems can only achieve the maximum throughput for either computation or memory bounded layers. Furthermore, specialized hardware units make one PIM architecture cannot be used to deploy multiple applications like CPUs. For example, the Transaction Validation MU cannot work as an EXP Translator [29], and the transposed inter block data transmission was only used during the CNN training process [23].

To improve the adaptability of current PIM systems, we propose to add several general purpose registers near the crossbar

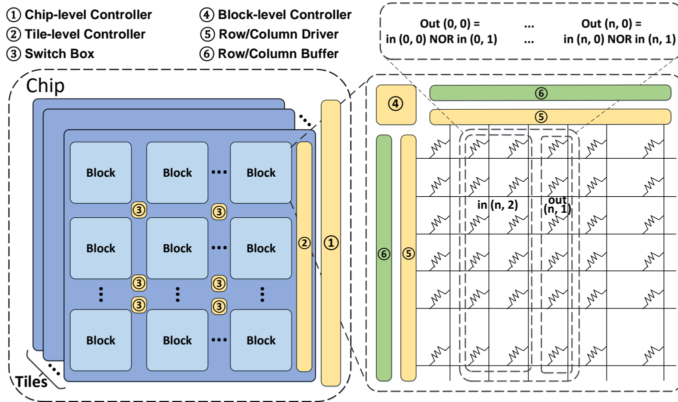


Fig. 4: System architecture.

logic and use instruction sets to control the dataflow. All PIM operations are abstracted as instructions, and one memristive memory block can serve as multiple-function units. When facing large datasets, more memory blocks can be assigned as buffers to reduce the on-chip off-chip data movement. When facing small datasets, higher throughput can be achieved by assigning more memory blocks as computation units.

Additionally, in this paper instead of implementing the naive FdGL implementation, we investigate acceleration strategies for quadtree/octree based FdGL implementation. We propose an Instruction Set Architecture (ISA) based general PIM solution for achieving the quadtree/octree. The quadtree/octree is widely applied in other visualization [45], high performance computing [13], and even machine learning systems [28]. As a result, our proposed PIM solutions can also be applied in these applications.

III. Methodology

In this section, we will elaborate on the design of the PIM system for the tree based FdGL implementation. Additionally, the dataflow of FdGL will be discussed. ISA and control logic supporting our system will also be introduced.

A. System Architecture

The architecture of our digital PIM system is based on memresitive memory cells with hierarchical control logic, as shown in Fig. 4. In contrast to other PIM designs [9], [29], [23], all memory blocks in our system are the same and the function of each block can be switched between storage and computation units flexibly. Software defined ISA is designed to support this PIM architecture, assigning where to fetch the data, which memory block will perform the arithmetic operations, and where to store the generated results. Different levels of control logic are responsible for instruction decoding, data fetching, and dataflow routing.

To better illustrate the workflow of our design, we will give an example of performing NOR operations in a row parallel way as shown in Fig. 4. For example, setting the voltage of the second and third column as V_0, the fourth column as GND, and leaving the voltage of other columns as $V_{isolate}$, can make the memory block perform NOR operations in each row of the second and third columns, and all of the generated results will be stored in the fourth column. If we only want to operate on the first row, the voltage of all other rows as $V_{isolate}$ should be adjusted through the column driver, then $out(0,0) = in(0,0)$ XOR $in(0,1)$ can be achieved.

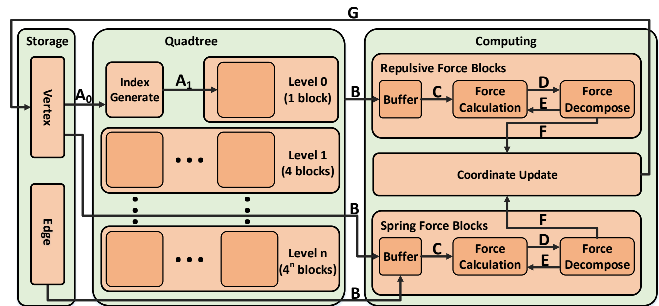


Fig. 5: Dataflow of FdGL.

B. FdGL Dataflow

In this section, we will discuss the dataflow of our tree based FdGL in the PIM system. We consider the FdGL as a data driven application and use SDF (Synchronous Data Flow [42]) graph to describe it. In traditional single PE (Processing Engine) architectures like a CPU, all nodes of the SDF graph will be mapped to one PE. After processing one node of the SDF graph, intermediate results are generated by this PE. These intermediate results will at first be transferred to off-chip memory, and then loaded back to on-chip memory for the computation of the next node in the SDF graph. Unlike in single PE architectures, in multiple PE architectures like systolic arrays and PIM systems, nodes of the SDF graph can be mapped to different PE units. Each PE unit represents one dependent state machine, so the intermediate results of each node are sent to the next node of the SDF graph directly without data movement between on-chip memory and off-chip memory. As a result, the dataflow optimization space in multiple PE architectures including PIM systems will be larger than single PE architectures, which will be explored in this section.

As shown in Fig. 5, we divide the whole SDF graph into three major parts and divide our PIM system into corresponding parts, which are the storage part, tree part and computing part. The computing part is responsible for computing the two kinds of forces and the updated coordinates of each vertex, the tree part is designed for maintaining the tree based data structure, and the storage part is reserved for storing vertex and edge information. In Fig. 5, each rectangular box represents one PE, and different PEs can work simultaneously. The basic unit in our PIM system is the memory block, and one PE can be consisted of one or multiple memory blocks. There will be intermediate results generated by the previous PEs, which are inputs for the subsequent PEs. We use letters between rectangular boxes to represent data transmissions between each PE.

To give a more intuitive view of the functionality of each PE, we choose to describe FdGL [36] in a way that usually

defines data driven applications (see algorithm 1). During the execution of FdGL, input data are fetched from $Vertex$ and $Edge$ blocks in the **Storage** part (A_0, B), then inserted into the **QuadTree** part based on the indexes calculated by $Index\ Generate$ blocks (A_1). This is described as Line 1 in algorithm 1. After constructing the quadtree, data will be sent to the **Computing** part (we use the same letter B since these data transmission can happen at the same time, See Line 3-4 in algorithm 1). Then the repulsive force (Line 6-8 in algorithm 1) and spring force (Line 10-13 in algorithm 1) will be calculated by $Force\ Calculation$ (C, D, E) blocks in the **Computing** part. After calculating the two kinds of forces, coordinates of each vertex will be updated by $Coordinate\ Update$ (F) blocks and sent back to the **Storage** part (G) (Line 15-16 in algorithm 1).

Algorithm 1 describes the dataflow of FdGL serially. But in real PIM systems some of the operations inside this algorithm can be processed in parallel. As Fig. 5 suggests, dataflow with the same label can be processed at the same time. To be more specific, Line 3 and Line 4 can be processed at the same time, which means transferring data from storage parts and tree node parts to computing parts can be done in parallel, shown as B in Fig. 5. The distance list calculation (Line 6 and Line 10), force calculation (Line 7 and Line 12), and force decomposition (Line 8 and Line 13) can also be processed in parallel, which is described as C, D, E in Fig. 5 respectively. After synchronizing two kinds of forces, F and G are performed to calculate the resultant force and update the coordinates.

To make full use of PIM resources, we also explored how to compute forces of different vertices in parallel, instead of just computing the forces acting on vertices one by one. To be more specific, In the loop of the algorithm 1 (Line 4), calculating forces acting on different vertices can be processed in parallel since there is no data dependency between each iteration of the loop. This data independence means the for loop can be fully unrolled as long as it meets the PIM resource requirement, and we define the unroll time as inter vertex parallelism. Higher inter vertex parallelism means the computing process of more vertices are done in parallel, which requires more PIM resources. For the FdGL of small graphs or other computing bounded applications, more PIM resources can be reserved for the computing part for a higher inter vertex parallelism. On the contrary, lower inter vertex parallelism is required to make our PIM system reserve more memory blocks for the intermediate results. With introducing the inter vertex parallelism, our solution can scale to fit graphs of different sizes onto our PIM systems.

C. Instruction Sets

In our system, we use two kinds of 64-bit instructions, PIM instructions and Memory instructions, to process various computing operations and memory operations in the FdGL algorithm. As shown in Fig. 6, PIM instructions consist of 7 parts, and Memory instructions consist of 5 parts. Among the 64 bits, bit 63 is used for differentiating PIM instructions from Memory instructions. Bit 62 of the PIM instruction represents the row/column flag, since operations inside PIM blocks can be processed in either a row parallel or column parallel way [26]. As for the opcode, PIM instructions support basic logic operations(e.g., NOR, NOT, $SHIFT$, etc.) and arithmetic operations (e.g., addition, subtraction, multiplication, division, etc.). Memory instructions support operations including read, write, and data transfer between different memory blocks. The $Row/Col\ Address$ represents the index of the row or column being operated on. Since each operation is processed in one row/column, only one row/column index is needed to specify that location. For memory instructions, to support data transfer operations, we need to differ the source and destination index, resulting in two row/column index fields in the instruction. The last three parts of PIM instructions represent the offset of first source operand, second source operand and destination operand inside the row/column. The last part of memory instructions represents the data size of the read, write and data transfer operations.

Algorithm 1 FdGL Dataflow on Digital PIM

Input: Coordinates of each Vertex; Edge information;
Output: Updated Coordinates of each Vertex; ▷ RF stands for Repulsive Force; SF stands for Spring Force

```
1: Reconstruct the quadtree T
2: for each tree node T_N in T do
3:     Retrieve all vertices v_list_r in T_N;
4:     Retrieve all edges in the graph connecting to node T_N;
5:     for each vertex v in v_list_r do
6:         Calculate distance lists dx_list_r, dy_list_r (v − ∗ in v_list_r);
7:         Calculate RF list f_list_r (v − ∗ in v_list_r);
8:         Calculate decomposed RF list fx_list_r, fy_list_r;
9:
10:        Calculate distance lists dx_list_s, dy_list_s (v − ∗ in v_list_s);
11:        ▷ v_list_s is the list of vertices having an edge connecting to v;
12:        Calculate SF list f_list_s (v − ∗ in v_list_s);
13:        Calculate decomposed SF lists fx_list_s, fy_list_s;
14:
15:        Calculate the sum of RF and SF fx, fy;
16:        Update the coordinate of v to (x, y);
17:    end for
18: end for
```

The compilation process for converting applications into our proposed ISA can be divided into two parts, precompilation and just-in-time compilation. When implementing FdGL, since the computation pattern is fixed and the graph information can be regarded as a preliminary, most of instructions are generated before the execution starts. After precompilation, the PIM system will start processing the input graph data. Just-in-time compilation happens during the execution. It will only be triggered during updating the quadtree/octree, since the tree nodes will be splitted only when reaching the maximum capacity. The additional ISA support for updating the tree based data structures will be discussed in Section III-E.

D. Control System

In this part, we will detail the functionality of each level of control logic in Fig. 4. ① is the chip level controller, the functionality of which is decoding instructions, distributes instructions to the destination tiles, and transfers data across

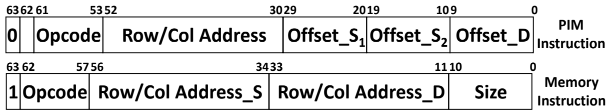


Fig. 6: Instruction format.

chips. ② is the tile level controller, which performs receiving instructions from the chip level controller and assigns it to the corresponding memory block. ③ represents the switching boxes, which interconnect blocks in the same tile and enables inter block data transmission. ④ is the last-level controller, which exists in every block. The function of ④ is to issue instructions at the last level of our memory system, which can be achieved by altering the voltage of the row and column driver. The row/column driver is denoted as ⑤ in Fig. 4.

The last level controller requires four additional general purpose registers for our tree based visualization system. In order to support operations run in a parallel way, controller ④ will not issue any instruction immediately after receiving it, and will wait for the next instruction to come and check if it is the same type. If they are the same type, the controller will just alter one more unit of the column or row driver. If they are not the same type, these accumulated previous instructions of the same type will be issued and performed as the vector processing. To support such batch processing, one register is required to store the current instruction type. Besides the instruction type register, three more registers are required for implementing tree based data structures. This will be discussed in Section III-E.

E. Tree Based Data Structure in the PIM System

As aforementioned, we discussed the ISA and control logic supporting the FdGL. But to implement the tree based optimized solution, additional hardware support is required to maintain the quadtree/octree data structure. This additional hardware support needs to have the ability to decide in which tree node each vertex should be inserted and detect if the tree node reaches the maximum capacity. To implement these functionalities, we propose to add three general purpose registers inside each memory block, which does not affect the generality of these tree blocks. In other words, these tree memory blocks can also be used for other functionalities if needed.

Before talking about the additional registers, we will show how to represent vertices and the quadtree/octree data structures in memory cells. As shown in Fig. 7, the length of each vertex is 128 bits. Bits 64-127 are used to represent the X-axis and Y-axis coordinates of each vertex in the 2D space. In the quadtree, each parent tree node has four child nodes, dividing the 2D space of that parent node into four sub parts. Bits 48-63 will be used to represent the index of each child node in the quadtree, and each level index requires two bits. The last bits 0-47 act as padding bits to make the total length of each vertex be a power of 2, and it can also act as reserved bits for the Z-axis coordinates when implementing an octree data structure, which can visualize graphs in a 3D space.

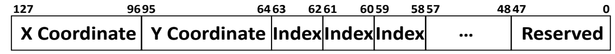


Fig. 7: Quadtree/Octree data structure format.

When implementing quadtree in PIM systems, we map each tree node to one physical memory block of the memristive chip. The information of each vertex in the same tree node is stored in the same memory block. Under this circumstance, inserting one vertex into the quadtree is similar to moving the 128-bit content from one memory block to another. However, there is still some difference between the "insert" instruction and "move" instruction. Based on our proposed instruction sets, the destination address of the "move" operation is part of the instruction, which means the routing of data transmission is decided by the "move" instruction itself. But for the "insert" operation, the final destination address cannot be determined merely by the instruction. The destination address has to be calculated during runtime. If the capacity of the destination tree node does not reach the maximum volume, the destination address is exactly the same as the value assigned by the "insert" instruction. Instead, the destination address has to be calculated during runtime and sent to the child nodes.

To make destination address able to be calculated at runtime, we split the "insert" instruction into 3 steps I_0, I_1, and I_2. I_0 is used to calculate the child node index of each level, which makes the final destination address calculation able to be achieved by simple addition and shift operations at runtime. After I_0 is finished, instruction I_1 can be issued and sent to the index generation block, then the padded 128-bit vertex information will be sent to the assigned tree node block referring to the destination address assigned by the instruction itself. There is one return bit of instruction I_1 to indicate if the capacity of that tree node reaches the maximum volume. If it is 1, the next instruction should be I_2, which is splitting that tree node and sending all vertices to the child tree node blocks.

To achieve these three steps, we need three more register R_1, R_2, and R_3, as shown in Fig. 8. After inserting one vertex into the assigned tree node, the counter register R_2 will be increased by 1. If it reaches the maximum volume, the status bit register R_3 will be set to 1, and returned to the host. Then the host will issue the I_2 instruction and set the valid bit register R_1 to 0. When transferring data from the tree node blocks to computing blocks, it will check the valid bit first, and only tree nodes with a valid bit 1 can send data to the computing nodes for the force calculations and coordinate updates.

Quadtree and octree can be implemented in our system because memristive cells can serve as both computation and storage units. However, since updating the quadtree is one part of the FdGL dataflow, we have to take the additional overhead of the "insert" instruction into consideration. There is a lot of interaction between the host device and the PIM chip when the three steps of the "insert" instruction are introduced. To amortise this overhead, we choose to use the double buffer technique, and to achieve this we deployed two quadtrees in

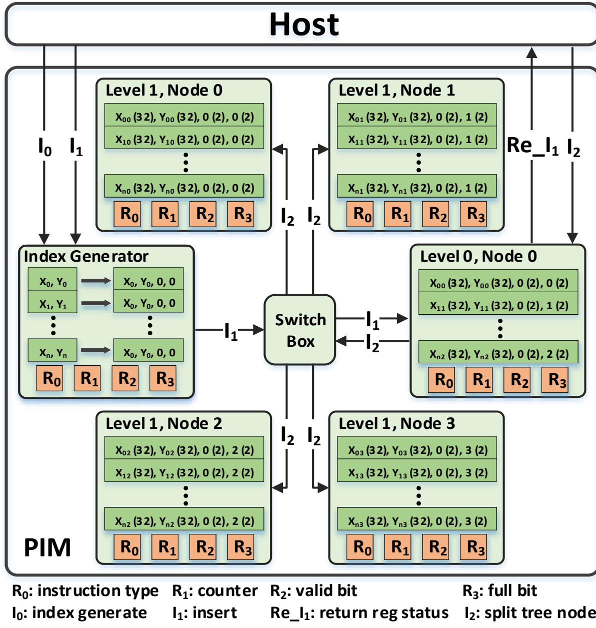


Fig. 8: QuadTree implementation.

our PIM system. One Vertex can be inserted to the quadtree for the next iteration as soon as its coordinates have been updated in the previous iteration. The prerequisite of the double buffer technique is that processing and updating the quadtree never becomes the bottleneck of the SDF graph. To prove this, we designed experiments in the evaluation part for measuring the processing time of each part.

IV. Implementation of Required Arithmetic Units

In this part, we will talk about the implementation of the arithmetic operations in FdGL, especially the division units. Moreover, natural log units and trigonometric units are also required for calculating spring forces and implementing force decomposition. All of these arithmetic units are implemented in 32-bit fixed point data precision, since the implementation of floating point addition requires additional hardware support [23], and the accuracy of the fixed point implementation is enough for our visualization system.

A. Division Units

Basic arithmetic units (addition and multiplication) are not enough to calculate the repulsive force. To do this, division units are also needed. In this part, we will show how to build division units efficiently on the PIM system. Based on the design of fixed-point multiplication units on PIM [7], we modify the division designs in CMOS platform [16] and propose the solution to achieve N fixed-point division in Algorithm 2. This implementation avoids involving too much control logic which is hard to be achieved in a row parallel way on PIM. Since the sign bit of the dividend and divisor can be easily calculated by XORing the sign bits of the dividend and divisor, we only consider the division of unsigned datatype in this situation. We assign the first N cells for the dividend, the next N cells for the divisor, and we reserve $2N$ cells for storing the quotient. Among the N bits, the first

Algorithm 2 Unsigned fixed-point division on digital PIM

Input: Total bits N, Integer bits N_I, Decimal bits N_D; Location of Dividend A $M_{0\sim N-1}$, Divisor B $M_{N\sim 2N-1}$; ▷ M_i denotes for the ith memory cell in the operated memory row
Output: Quotient is stored in location $M_{2N\sim 4N-1}$;

```
1:  M_{5N~6N-1} ← M_{0~N-1};
2:  M_{8N~9N-1} ← M_{N~2N-1};
3:  M_{7N+1~8N-1} ← M_{8N~9N-1};
4:  for each i ∈ [0, 2N-1] do
5:      M_{10N~13N-1} ← SUB(M_{4N~7N-1}, M_{7N~10N-1});
6:      update_dividend(i);
7:      M_{2N+i} ← M'_{10N};
8:      M_{7N+1~8N-1} ← M_{7N+1+i~8N+i};
9:  end for
10:
11: function UPDATE_DIVIDEND(i):                    ▷ Version 1
12:     if M_{10N} == 0 then
13:         M_{4N~7N-1} ← M_{10N~13N-1};
14:     else
15:         M_{4N~7N-1} ← M_{4N~7N-1};
16:     end if
17: end function
18:
19: function UPDATE_DIVIDEND(i):                    ▷ Version 2
20:     for each j ∈ [0, 3N-1] do
21:         M_{13N+j} ← M'_{10N};
22:         M_{16N+j} ← M_{10N};
23:     end for
24:     M_{4~7N-1} ← (M_{4~7N-1} & M_{13~16N-1}) || (M_{10~13N-1} & M_{16~19N-1});
25: end function
```

N_I bits are integer bits and the other N_D bits are decimal bits. Our proposed algorithm 2 is defined by the equation $Dividend = Divisor \cdot \sum_{i=-N}^{N-1} (2^i \cdot Quotient_i) + Reminder$, which will get the quotient bit by bit and can be achieved in a row parallel way.

First, we make a copy of the dividend and divisor (Line 1-2) for iterative subtraction, and reserve $3N$ bits for left and right shifting. The division unit will calculate the quotients from the highest to the lowest bit, and start left-shifting the divisor by $N-1$ bits (Line 3). Then the value of $Divisor$ is updated to $2^{N-1} \cdot Divisor$, which will be used to compare with the dividend to decide whether the highest bit of the quotient is 1 or 0. The other $2N-1$ bits of the quotient are calculated by right shifting the divisor bit by bit (Line 4-9). If the quotient bit is 1 instead of 0, the dividend will be updated to $Dividend - Divisor$ (Line 11-17). If the divisor is larger than the dividend, which means the quotient bit is 0, the dividend will not be updated. To avoid comparison operations inside the memristive cells, we propose to use logic operations (Version 2 of function Update_dividend) instead of conditional branches (Version 1 of function Update_dividend) to update the dividend.

B. Implementation of Other Arithmetic Units

In addition to the division units supporting repulsive forces, natural log units are required to implement the spring force. Currently, there are many efficient hardware solutions for natural log units [32], [37], and most of them are based on Taylor Series. The Taylor Series of $\ln x$ on $x = 1$ can be described as equation 4a. The prerequisite of the equation is $x \in (1, 2)$, but we need to extend it to the whole real number domain. Note that each real number val can be expressed in the form of $val = 2^{exp} \cdot (1.mantissa)$, in which exp is an

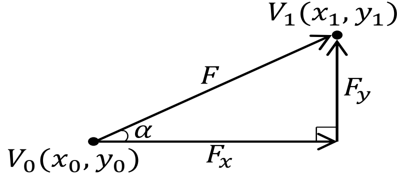


Fig. 9: Force decomposition.

integer, and each $(1.mantissa)$ can represent any real number in the range of $(1, 2)$. In this way, the natural log value of each real number val can be expressed as equation 4b. The implementation of $\ln 2 \cdot exp$ relies on multiplication units, and the value of $\ln(1.mantissa)$ is obtained by equation 4a. All of these arithmetic operations can be processed in a row parallel way, so the natural log units can be implemented efficiently in the PIM system.

$$\ln x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}(x-1)^n}{n} \;\; (1 < x < 2) \tag{4a}$$

$$\ln(val) = \ln 2 \cdot exp + \ln(1.mantissa) \tag{4b}$$

To obtain the resultant forces, we need to decompose each repulsive force and spring force and calculate the component of the X and Y axes, which requires trigonometric units or other equivalent units. As shown in Fig. 9, the force F (node V_1 acts on V_0) can be decomposed to F_x and F_y, in which $F_x = F \cdot \cos\alpha$, $F_y = F \cdot \sin\alpha$. $\sin\alpha$ can be expressed as $\sin\alpha = \sin\arctan\frac{y_1 - y_0}{x_1 - x_0}$, so the force decomposition units can be calculated with a combination of $\arctan$ units, $\sin$ units, and $\cos$ units. To obtain a more flexible and efficient design, we choose to use Taylor Series for implementing these trigonometric units in a similar way as the natural log units.

V. Analytical Estimation

In this section, we conduct a preliminary analytical evaluation of CPUs, GPUs, and the proposed PIM architecture. At first, we quantify the arithmetic operations and data movement inside the workload. Then we calculate the performance on CPU, GPU and PIM systems in a uniform manner based on the maximum parallelism and memory bandwidth of each system.

A. Workload Analysis

In this paper, we are using six real graphs as benchmarks shown in table II, and we set the constant parameters to $C_1 = 2$, $C_2 = 1$, $C_3 = 1$, $iteration_time = 100$ [6]. We use prior work [6] as the naive FdGL implementation (Implementation I), and use work [36] as the quadtree based optimized implementation (Implementation II). To better understand the arithmetic operation reduction on quadtree based implementation and additional overhead introduced by the quadtree, we breakdown the FdGL into four parts, and show the results in table III.

We consider one addition or multiplication as one arithmetic operation, and the square root, log unit arithmetic operation is referred from prior works [43], [27]. In the quadtree based implementation II, the arithmetic operation reduction comes from the repulsive force, and the arithmetic operations for spring force and coordinate update remains the same. Note that based on different algorithm implementations, the arithmetic operations can be slightly different, but in general the time complexity remains the same.

To model the generated intermediate results and calculate the required memory bandwidth, we divide the intermediate results into three parts, intra-loop data, inter-loop data, and external data. Intra-loop data are usually the results of previous arithmetic operations and will be the operands for the subsequent operations. The intra-loop data have no temporal locality, so they are stored in registers. Inter-loop data will be used in next loops, so we use cache to store them. External data are intermediate results that have to be moved between on-chip and off-chip memory, like data movement when updating quadtree. The overhead of the quadtree based implementation comes from the additional inter-loop and external data.

B. Hardware Configuration

We will evaluate the performance of FdGL on one CPU platform (Xeon(R) Platinum 8160), two GPU platforms (Nvidia GTX 1080Ti and Tesla V100), and our proposed PIM architecture. We use an additional GPU_IT platform which provides the same throughput as the PIM platform for calculating the potential performance provided by PIM architecture. The memory bandwidth on the GPU_IT platform is also adjusted based on the ratio of throughput to memory bandwidth on Tesla V100. The hardware configuration details are shown in table I. For the theoretical maximum throughput of the CPU platform, we assume all cores run at the maximum frequency with a perfect pipeline design, which means each arithmetic operation only takes one clock cycle on average. The maximum throughput for the two GPU platforms are referred from Nvidia white books [5], [4]. For the PIM architecture, we choose 1,024 bit as the raw size corresponding to the 32-bit operands [7], [23]. The throughput for our PIM architecture is calculated based on the maximum parallelism ($8GB/1,024b = 64M$) and arithmetic operation latency from prior work [7].

C. Performance Estimation

In this part, we will calculate the theoretical performance on CPU, GPU and PIM platforms, showing the potential throughput improvement provided by PIM architecture. The overall execution time can be divided into two parts, one is the time for arithmetic operations, the other is the time for data movement between off-chip and on-chip memory. The execution time for arithmetic operations is decided by the number of arithmetic operations and throughput of the hardware platforms, which can be calculated directly with the data in table I, II, and III. However, in real cases CPUs and GPUs cannot achieve the maximum throughput due to the limited register capacity. We define the variable coefficient of utilization to represent the ratio of real throughput to maximum throughput. This ratio is decided by the amount of intra-loop data and register capacity. The execution time for data movement is decide by the scale of generated inter-loop and external data and the DRAM bandwidth. With the

TABLE I: Hardware configurations.

Platform	CPU	GPU	GPU	GPU	PIM
Name	Xeon(R) Platinum 8160	GTX 1080Ti	Tesla V100	Iso-Throuhput (IT)	N/A
Frequency	3,700MHz	1,530MHz	1,582MHz	816MHz	800 MHz
Cache	L1 1536KB	Reg 7168 KB	Reg 20480 KB	Reg 20480 KB	N/A
	L2 24MB	L2 2816 KB	L2 6144 KB	L2 6144 KB	
	L3 33MB				
Memory	192GB GDDR4	11GB GDDR5X	16GB HBM2	16GB HBM2	8GB
DRAM Bandwidth	250GB/s	484GB/s	900GB/s	477GB/s	N/A
(CUDA) Cores	24	3,584	5,120	5,120	N/A
Throuphput	178 GFLOPS	11.3 TFLOPS	15.7 TFLOPS	8.1 TFLOPS	8.1 TFLOPS

TABLE II: Benchmarks.

Benchmark	Musae-Twitch-PT	Musae-Twitch-RU	Musae-Twitch-EN	Musae-Facebook	Email-Enron	Musae-Github
Abbreviation	PT	RU	EN	FB	Enron	Git
# of Nodes	1,912	4,385	7,126	22,470	36,692	37,700
# of Edges	31,299	37,304	35,324	171,002	183,831	289,003
Density	0.017	0.004	0.002	0.001	0.001	0.001

help of cache on CPU and GPU platforms, not all of the inter-loop data have to be transferred between off-chip and on-chip memory. Under this circumstance, we introduce another variable cache miss rate to model the data movement time.

Fig. 10 shows the average normalized execution time of the six graphs on CPU and GPU platforms under different coefficient of utilization and cache miss rates. We use CPU_I, GTX_I, V100_I, and GPU_IT_I to represent the results of implementation I on different hardware platforms, and we use CPU_II, GTX_II, V100_II, and GPU_IT_II to represent the results of implementation II. CPU cannot perform as well as GPU and PIM platform due to the limited parallelism. Because GPU platforms have higher arithmetic IPC (instructions per cycle) than PIM, our proposed PIM architecture can only achieve a $1.44\times$ and $1.03\times$ throughput on arithmetic operations as GTX_II and V100_II when the coefficient of utilization is 0.5. However, the advantage of PIM system is the removal of data movement. When the coefficient of utilization and cache miss rates are 0.3 and 0.7, our PIM architecture can achieve a $11.38\times$, $6.56\times$ and $8.07\times$ throughput as the GTX_II, V100_II, and GPU_IT_II solution respectively.

D. Limitations of the Analytical Model

Based on our calculation, the PIM architecture can achieve at most $11.38\times$ throughput as the GPU platform. However, this result is based on our assumptions of the coefficient of utilization and cache miss rates. In real platforms, the coefficient of utilization rate is also affected by thread contention and data synchronization, which cannot be obtained based on calculation. As for the data movement, the maximum memory bandwidth can only be achieved when transferring data from contiguous DRAM address to on-chip memory. In addition, we did not consider the data movement between different levels of cache in CPU and GPU platforms, and the data movement between different memory blocks in PIM systems. To obtain more accurate results, we will design experiments on real CPU and GPU platforms and a cycle-accurate PIM simulator in Section VI.

VI. Experimental Evaluation

In this section, we will evaluate the efficacy of our proposed PIM architecture. We run the FdGL on real CPU, GPU platforms and a cycle-accurate PIM simulator to justify the performance improvement on PIM systems. We also evaluate the energy consumption reduction on PIM systems and the scalability of our design. The accuracy loss introduced by PIM systems will also be discussed in this section.

TABLE III: Breakdown of FdGL operations.

		Implementation I	Implementation II (Quadtree based)
Repulsive Force	Arithmetic Operations	**20 * (|V|-1) * |V| / 2 * N**	**(20 * (T-1) * |V| / 2 + 4 * H * |V|) * N**
	Intra-Loop Data	**13 * (|V|-1) * |V| / 2 * N**	**13 * (T-1) * |V| / 2 * N**
	Inter-Loop Data	**2 * |V| * N**	**(T * |V| + 2 * |V|) * N**
	External Data	**2 * |V|**	**0**
Spring Force	Arithmetic Operations	23 * |E| * N	23 * |E| * N
	Intra-Loop Data	17 * |E| * N	17 * |E| * N
	Inter-Loop Data	0	0
	External Data	0	0
Coordinate Update	Arithmetic Operations	4 * |V| * N	4 * |V| * N
	Intra-Loop Data	2 * |V| * N	2 * |V| * N
	Inter-Loop Data	2 * |V| * N	2 * |V| * N
	External Data	2 * |V|	2 * |V|
Quadtree Update	Arithmetic Operations	N/A	**4 * H * |V| * N**
	Intra-Loop Data	N/A	**2 * H * |V| * N**
	Inter-Loop Data	N/A	**$2 * 4^{H-1} * T * N$**
	External Data	N/A	**4 * |V| * N**

$|V|, |E|$ represents number of Vertices, Edges. T, H, N represents tree node volume, tree height, number of iterations.

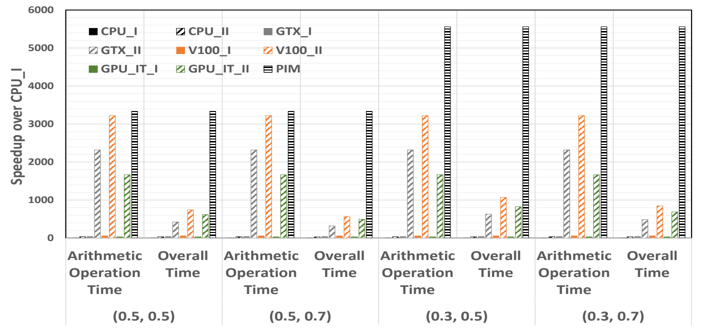


Fig. 10: Analytical estimation of speedups.

A. Evaluation Setup

We evaluate our design utilizing the experimental methodology in [23] after enhancing the FloatPIM simulator to include division and natural log units. We choose the same PIM configuration as the FloatPIM [23]. The capacity of the PIM chip is 8Gb, consisting of 32 memory tiles. Each memory tile consists of 256 crossbar memory blocks, both the column and row size of which are 1,024. Various circuit parameters are referenced from FloatPIM [23] and Pipelayer [21]. The latency and energy consumption of the basic logic operation are 1.1ns and 0.29fJ, and the read/write latency and read/write energy are 29.31ns/50.88ns and 1.08pJ/3.91nJ, respectively. We keep the Switches and Controller of the FloatPIM design, and the power of them are 0.42mW and 0.65mW. Unlike in FloatPIM, we discard the Shifter and Max Pool, since they are designed for floating-point addition operations and pooling layers in CNNs. To support our proposed ISA, We add additional decode logic inside the block-level controllers.

We use Intel's RAPL [19] tool and Nvidia-SMI [3] tool to collect the energy consumption on CPU and GPU platforms. We use Synopsys PrimeTime [1] to measure the power of our decoder logic, which is 0.26mW. We also add 4 general purpose registers inside each memory block to support our quadtree data structures. The overhead of introducing addi-

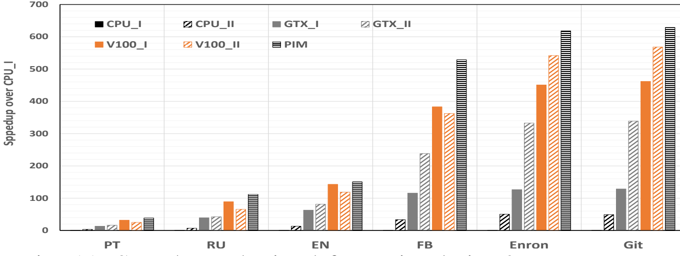


Fig. 11: Speedups obtained from simulation&measurement.

tional registers is measured using the same methodology as the PIM operations in our simulator because these 4 general purpose registers can be implemented by memristive cells.

B. Performance Comparison with CPU&GPU Platforms

We compare the performance of our PIM design to the CPU and GPU platform designs, and the results are shown in Fig. 11. Our PIM system achieves a $346.18\times$ performance improvement compared to the CPU Implementation I and $13.33\times$ performance improvement over the CPU Implementation II. Additionally, our PIM design performs as well as the optimized GPU design, obtaining an average performance increase of $2.14\times$ over the Nvidia GTX 1080Ti platform, and $1.37\times$ over the Tesla V100 platform, which are two of the most powerful commercial GPU platforms.

We present the execution time of arithmetic operations and data movement in real hardware platforms in Table IV. Different from our analytical model, the data movement in real platforms consists of inter-platform and intra-platform data movement. Inter-platform data movement represents the data movement between host device and GPU or PIM on-chip memory. For the GPU Implementation II, in which quadtree is introduced to prune the computation, the update process of quadtree tree on host device is also considered as one part of data movement overhead. For both GPU and PIM platforms, there is still data movement inside the platform itself. For example, in GPU platform data have to be transferred between each block for synchronization, while in PIM platform data are transferred from one memory block to the other acting as the a synchronous dataflow. Such kinds of data movement contribute to intra-platform data movement.

Compared with GPU Implementation I, our PIM solution performs better due to the reduction of computation. When quadtree is introduced, a huge proportion of the computation is pruned, and the time complexity is decreased from $O(|V|^2)$ to $O(|V|\log|V|)$. Our PIM solution achieves better performance than GPU Implementation II because of the removal of inter-platform data movement overhead. In GTX_II, the data movement contributes 93.19% to the overall executing time, and in V100_II the ratio is 95.09%.

C. Energy Comparison with CPU&GPU Platforms

In this section, we discuss the energy savings of our PIM platform over CPU and GPU platforms and show results in Fig. 12. Due to its high generality and complicated control logic, the energy consumption of the CPU Implementation I is $1931.05\times$ higher than the PIM platform. Even though the time

TABLE IV: GPU&PIM experimental time breakdown.

Platform	Breakdown (ms)	PT	RU	EN	FB	Enron	Git
GTX_I	Arith Oper Time	185.65	171.42	188.50	199.10	202.77	204.28
	Data Move Time	774.35	1398.58	2331.50	13550.90	32927.23	34685.72
GTX_II	Arith Oper Time	143.98	149.97	156.15	277.69	469.06	420.91
	Data Move Time	856.02	1530.03	2093.85	6792.31	12560.94	13309.09
V100_I	Arith Oper Time	140.41	158.87	168.59	211.17	225.14	219.85
	Data Move Time	269.59	531.13	941.41	3978.83	9124.86	9510.15
V100_II	Arith Oper Time	49.62	62.42	48.26	70.19	182.09	276.98
	Data Move Time	390.38	817.58	1301.74	4229.81	7447.91	7623.02
PIM	Arith Oper Time	96.22	313.66	371.63	1388.52	2250.22	2267.97
	Data Move Time	209.38	237.02	681.12	1626.37	4541.95	4871.17

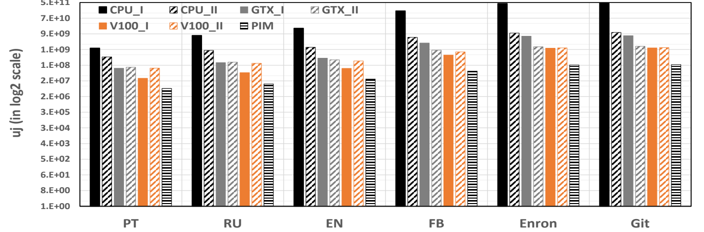


Fig. 12: Energy comparison of PIM with CPU & GPU.

complexity is decreased from $O(|V|^2)$ to $O(|V|\log|V|)$, the energy of CPU Implementation II is still $74.51\times$ higher than our PIM system. The computational power of the the GPU platforms will also result in higher energy consumption. In this situation, our design achieved $30.18\times$ and $6.55\times$ energy savings over GPU Implementation I, and $14.30\times$, and $12.16\times$ energy savings over the quadtree based GPU Implementation II on the 1080Ti and V100 platforms respectively.

D. Breakdown of the FdGL SDF graph

Our PIM platform achieves better performance and lower energy consumption than the CPU and GPU platforms because of our advanced scheduling system for the FdGL dataflow. This section breaks down the processing time of each node in the FdGL SDF graph to investigate a more efficient degree of inter vertex parallelism as well as a more suitable memory capacity parameter for our system.

As aforementioned, to utilize the PIM resources more efficiently, in addition to fully unroll the force computation of one vertex (Line 5-7, 9-11 in Algorithm 1), the inter vertex parallelism is also supposed to be taken into consideration. This refers to the unrolling time of the Loop (Line 4 in Algorithm 1). Allocating more hardware resources to increase the parallelism of the force computation will result in a better performance. As for the cost, the data transfer bandwidth requirement inside the PIM chip will also increase, and because of this we cannot assign hardware resources arbitrarily and have to investigate the appropriate level of parallelism. Our goal is to balance the processing time of each node of the SDF graph in Fig. 5, in order to balance the data transfer bandwidth with the inter vertex parallelism requirement.

In Fig. 13, we show the proportion of the processing time of each node in the FdGL SDF graph, and we select 32, 64, 128, and 256 as the degree of inter vertex parallelism. A represents updating the quadtree. In all cases the processing time of stage A is not the most time consuming, making the double buffer technique mentioned in Section III-E realistic. Stages C, F, and G perform most of the data synchronization and vertex

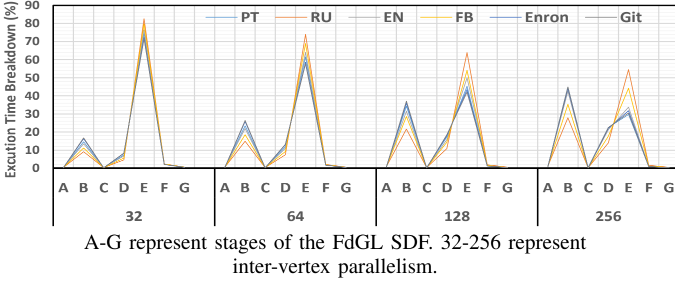


Fig. 13: Breakdown of the SDF Dataflow.

coordinate update jobs, which are not computing intensive, and consume less processing time and hardware resources than force computation. For the other three nodes, when the degree of inter vertex parallelism is low, E which performs force computation contributes most of the processing time. As parallelism increases from 32 to 256, the processing time of B, D which perform data transfer will contribute more to the overall processing time. To avoid making data transfer bandwidth inside PIM chips the bottleneck of our system, we choose not to increase parallelism and set 256 as the optimal solution. A lower degree of inter vertex parallelism means a lower memory capacity requirement, allowing our PIM system to support memristive chips with smaller capacity. Higher inter vertex parallelism means the 8Gb memory capacity can be fully utilized, achieving a better performance.

E. Accuracy

Our PIM design brings a significant performance improvement and energy reduction over CPU and GPU implementations, but there is still a small accuracy loss caused by our fixed point or approximate implementation of arithmetic units and the introduce of quadtree.

1) Definition of Accuracy: The accuracy loss of our system comes from two different sources, one is from the algorithm level, while the other is from the hardware implementation. Because of the introduction of quadtree, only vertices in the same tree node exert the repulsive force on each other instead of vertices in the whole graph, which results in an accuracy loss. At the hardware level, the fixed point data precision will also contribute to a small amount accuracy loss. For the implementation of trigonometric and natural log units, we choose to use Taylor Series, which computes the approximate rather than the exact values of trigonometric functions and will also contribute to the accuracy loss.

We choose the Implementation I on the CPU platform with 32-bit floating point data precision as the baseline. We use the absolute error instead of relative error to calculate the accuracy loss. We are visualizing the six benchmarks in a 2D space of $(range, range)$. The coordinates of each vertex have no real meaning other than location. In other words, one vertex moving from $(1, 1)$ to $(2, 2)$ will contribute equally to the total accuracy loss as another vertex moving from $(100, 100)$ to $(101, 101)$. We define the global accuracy loss as: $loss = \sum_{i=1}^{n} \frac{\sqrt{(\bar{x}_i - x_i)^2 + (\bar{y}_i - y_i)^2}}{n*range}$, where n represents the number of vertices, $(\bar{x}_i, \bar{y}_i)$ represents the final coordinate of vertex i of the CPU baseline, and (x_i, y_i) represents the final coordinate of vertex i of our PIM implementation.

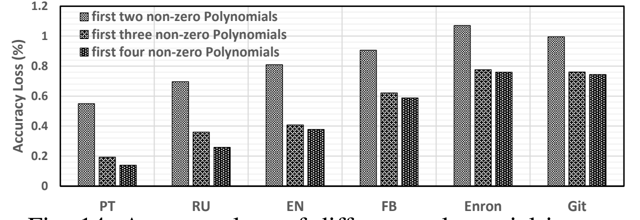


Fig. 14: Accuracy loss of different polynomial items.

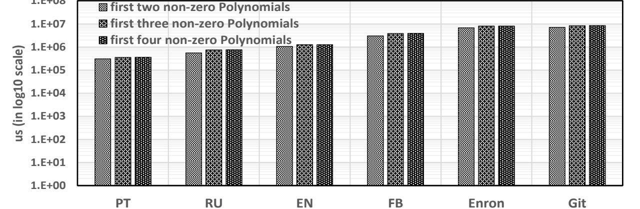


Fig. 15: Processing time of different polynomial items.

2) Trade-off between Accuracy and Performance: As discussed in Section IV, we choose to use Taylor Series to implement the trigonometric and log units. To balance accuracy and throughput, we have to choose the appropriate number of non-zero items in our Taylor Series. If only a few items are applied, the accuracy loss might be unacceptable, however if we apply too many items, the overall performance can be affected.

As Fig. 14 suggests, the accuracy of the six benchmarks follow the same trend. The more non-zero elements in the polynomial, the more accurate results our PIM design can obtain. More non-zero terms in our polynomial also result in a longer processing time (shown in Fig. 15). When increasing the number of non-zero terms from 2 to 3, processing time increases by 22.27% while accuracy increases by 0.31%. However from 3 to 4, processing time increases by 1.08% while accuracy only increases by 0.04%. When the number of non-zero terms is larger than 3, the marginal accuracy gain of additional polynomial terms is insignificant. Thus, we choose to use top 3 non-zero terms to implement our Taylor Series for the trigonometric and log units.

3) Overall Accuracy Loss: Aside from the accuracy loss introduced by trigonometric and log units, the fixed point data precision [22] and quadtree based acceleration strategy [45] could also contribute to the accuracy loss. As Fig. 16 shows, the average accuracy loss of the six graphs is 0.65%. There is a accuracy loss discrepancy between each graph, and the reason is that we are visualizing the six graphs with a different number of vertices and edges in the same 2D space. If a large graph is visualized in a small space, it will result in a closer average distance between each vertex. Once the distance between two vertices approaches zero, the repulsive force and spring force between them will be infinity. As a result, being closer to infinity means more accuracy loss. In real cases, the range of 2D or 3D spaces for visualizing graphs should be proportional to the scale of the graphs. Thus, the little accuracy discrepancy in our experiments will be eliminated.

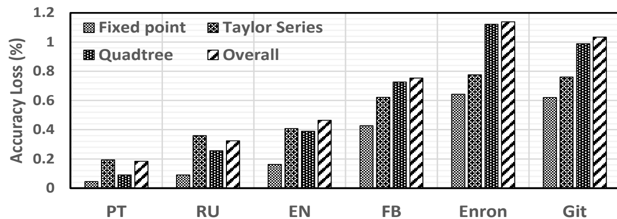


Fig. 16: Accuracy loss.

VII. Conclusion

In this paper, we propose a novel PIM architecture to accelerate a graph visualization algorithm. We design instruction sets as well as logic for implementing arithmetic operations and maintaining a tree based data structure; the flexible ISA that we created make our PIM system fit for datasets of different sizes. Moreover, we fully explore the dataflow of the algorithm and develop a synchronous dataflow based PIM system to deploy it. Analytical estimation shows an $8.07\times$ performance improvement if our proposed PIM architecture have the same maximum throughput as GPU platforms. Experimental results of 6 real graph datasets show that our PIM solution significantly outperforms state-of-the-art CPU and GPU systems, yielding $13.33\times$ and $2.14\times$ speedups, $74.51\times$, and $14.30\times$ energy savings over optimized CPU and GPU solutions, respectively.

Acknowledgement: This research was supported in part by NSF grant numbers 1725743, 1745813, and 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 National Science Foundation or other sponsors.

References

[1] Himanshu Bhatnagar. *Advanced ASIC chip synthesis*. Springer, 2002.
[2] Nvidia Corporation. Nvidia cuda toolkit documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html/, 2019.
[3] Nvidia Corporation. Nvidia system management interface. https://https://developer.nvidia.com/nvidia-system-management-interface/, 2020.
[4] Nvidia Corporation. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2020.
[5] Nvidia Corporation. Nvidia turning gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2020.
[6] P. Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.
[7] A. Haj-Ali et al. Efficient algorithms for in-memory fixed point multiplication using magic. In *ISCAS*, 2018.
[8] A. Panagiotidis et al. Consistently gpu-accelerated graph visualization. In *VINCI*, 2015.
[9] A. Shafiee et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.
[10] B. Jenny et al. Force-directed layout of origin-destination flow maps. *International Journal of Geographical Information Science*, 31(8):1521–1540, 2017.
[11] B. Schwikowski et al. A network of protein–protein interactions in yeast. *Nature biotechnology*, 18(12):1257–1261, 2000.
[12] Benedek Rozemberczki et al. Multi-scale attributed node embedding, 2019.
[13] C. Burstedde et al. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.
[14] C. Mueller et al. Distributed force-directed graph layout and visualization. *EGPGV*, 6:83–90, 2006.
[15] C. Zhang et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[16] D. A. Patterson et al. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.
[17] D. Chakrabarti et al. R-mat: A recursive model for graph mining. In *SDM*, 2004.
[18] D. Holten et al. Force-directed edge bundling for graph visualization. In *Computer graphics forum*, 2009.
[19] H. David et al. Rapl: memory power estimation and capping. In *ISLPED*, 2010.
[20] I. Herman et al. Graph visualization and navigation in information visualization: A survey. *TVCG*, 6(1):24–43, 2000.
[21] L. Song et al. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.
[22] M. Imani et al. Digitalpim: Digital-based processing in-memory for big data acceleration. In *GLSVLSI*, 2019.
[23] M. Imani et al. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA*, 2019.
[24] M. LeBeane et al. Data partitioning strategies for graph workloads on heterogeneous clusters. In *SC15*, pages 1–12, Nov 2015.
[25] N. Quinn et al. A forced directed component placement procedure for printed circuit boards. *IEEE Transactions on Circuits and systems*, 26(6):377–388, 1979.
[26] N. Talati et al. Logic design within memristive memories using memristor-aided logic (magic). *IEEE Transactions on Nanotechnology*, 15(4):635–650, 2016.
[27] O. Vinyals et al. A hardware-independent fast logarithm approximation with adjustable accuracy. In *ISM*, 2008.
[28] P. Wang et al. O-cnn: Octree-based convolutional neural networks for 3d shape analysis. *ACM Transactions on Graphics (TOG)*, 36(4):1–11, 2017.
[29] Q. Wang et al. Re-tangle: A reram-based processing-in-memory architecture for transaction-based blockchain. In *ICCAD*, 2019.
[30] R. Davidson et al. Drawing graphs nicely using simulated annealing. *TOG*, 15(4):301–331, 1996.
[31] S. C. Teja et al. Power network layout generation using force directed graph technique. In *NPSC*, 2014.
[32] S. J. Melnikoff et al. Implementing log-add algorithm in hardware. *Electronics Letters*, 39(12):939–940, 2003.
[33] S. Song et al. Proxy-guided load balancing of graph processing workloads on heterogeneous clusters. In *ICPP*, pages 77–86, Aug 2016.
[34] S. Song et al. Start late or finish early: A distributed graph processing system with redundancy reduction. *Proc. VLDB Endow.*, 12(2):154–168, October 2018.
[35] T. Kamada et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.
[36] T. MJ. Fruchterman et al. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.
[37] W. Wong et al. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *TC*, 43(3):278–294, 1994.
[38] X. Zhu et al. Gemini: A computation-centric distributed graph processing system. In *OSDI 16*, pages 301–316, GA, 2016. USENIX Association.
[39] Y. Wang et al. Two improved gpu acceleration strategies for force-directed graph layout. In *ICCASM*, 2010.
[40] Y. Zhang et al. Pattpim: A practical reram-based dnn accelerator by reusing weight pattern repetitions. In *DAC*, 2020.
[41] Y. Zhuo et al. Graphq: Scalable pim-based graph processing. In *MICRO*, 2019.
[42] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
[43] Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, 32, 2003.
[44] Stanford. The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/, 2014.
[45] Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.
[46] Xilinx. Xilinx products overview. https://www.xilinx.com/products/silicon-devices/fpga.html, 2020.