Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators

Marzieh Lenjani
Marzieh.Lenjani@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Alif Ahmed
Alifahmed@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

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

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

ABSTRACT

Processing-in-memory (PIM) minimizes data movement overheads by placing processing units near each memory segment. Recent PIMs employ processing units with a SIMD architecture. However, kernels with random accesses, such as sparse-matrix-dense-vector (SpMV) and sparse-matrix-sparse-vector (SpMSpV), cannot effectively exploit the parallelism of SIMD units because SIMD's ALUs remain idle until all the operands are collected from local memory segments (memory segment attached to the processing unit) or remote memory segments (other segments of the memory).

For SpMV and SpMSpV, properly partitioning the matrix and the vector among the memory segments is also very important. Partitioning determines (i) how much processing load will be assigned to each processing unit and (ii) how much communication is required among the processing units.

In this paper, first, we propose a highly parallel architecture that can exploit the available parallelism even in the presence of random accesses. Second, we observed that, in SpMV and SpMSpV, most of the remote accesses become remote accumulations with the right choice of algorithm and partitioning. The remote accumulations could be offloaded to be performed by processing units next to the destination memory segments, eliminating idle time due to remote accesses. Accordingly, we introduce a dispatching mechanism for remote accumulation offloading. Third, we propose *Hybrid partitioning* and associated hardware support. Our partitioning technique enables (i) replacing remote read accesses with broadcasting (for only a small portion of data that will be read by all processing units), (ii) reducing the number of remote accumulations, and (iii) balancing the load.

Our proposed method, Gearbox, with just *one* memory stack, delivers on average (up to) 15.73× (52×) speedup over a server-class GPU, NVIDIA P100, with *three* stacks of HBM2 memory.

ISCA '22, June 18–22, 2022, New York, NY, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527402

CCS CONCEPTS

• **Computer systems organization → Special purpose systems.**

KEYWORDS

PIM, SpMV, SpMSpV, sparse, processing in memory, graph

ACM Reference Format:
Marzieh Lenjani, Alif Ahmed, Mircea Stan, and Kevin Skadron. 2022. Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527402

1 INTRODUCTION

In current computing systems, the latency and energy consumption of fetching data from off-chip memory can be 2-3 orders of magnitude higher than an arithmetic operation [23]. Processing-in-memory (PIM) architectures alleviate this data movement overhead by placing processing units near memory segments (banks or subarrays) [21, 30, 32, 35].

SpMV and SpMSpV are essential computational kernels that are widely used and *memory-intensive* (requiring few computations per loaded datum from memory). The generalized forms of SpMV and SpMSpV, where the multiplication and addition can be replaced by other operations, appear in many important application domains such as machine learning (e.g., Support Vector Machine and Sparse K-Nearest Neighbor) and graph processing (e.g., Page Rank) [9, 41, 42].

Due to SpMV and SpMSpV kernels' memory-bound nature and widespread applications in various domains, they are natural candidates for PIM acceleration. Adding support for these kernels to PIM-based accelerators can boost such applications' performance, expand the market for PIM, and increase vendors' motivation in PIM investment.

However, existing PIM architectures often are only optimized for regular kernels by providing high parallelism using SIMD units [24, 30] or bit-level parallelism [21, 35]. In this paper, we introduce a PIM architecture that provides high parallelism for SpMV and SpMSpV. Later, we demonstrate that our proposed architecture outperforms SIMD approaches for regular kernels as well.

There are two major approaches for SpMV and SpMSpV: i) row-oriented or matrix-driven approach (Figure 1(a)), and ii) column-oriented or vector-driven approach (Figure 1(b)). The row-oriented

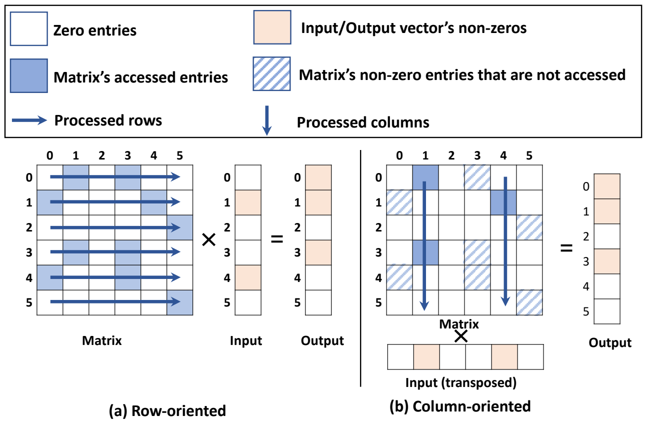


Figure 1: The row-oriented approach processes all rows, whereas the column-oriented approach processes only the columns corresponding to the non-zero entries of the input vector. In (b), the input vector is transposed to illustrate the relation between non-zero entries of the input vector and the processed (activated) columns of the matrix.

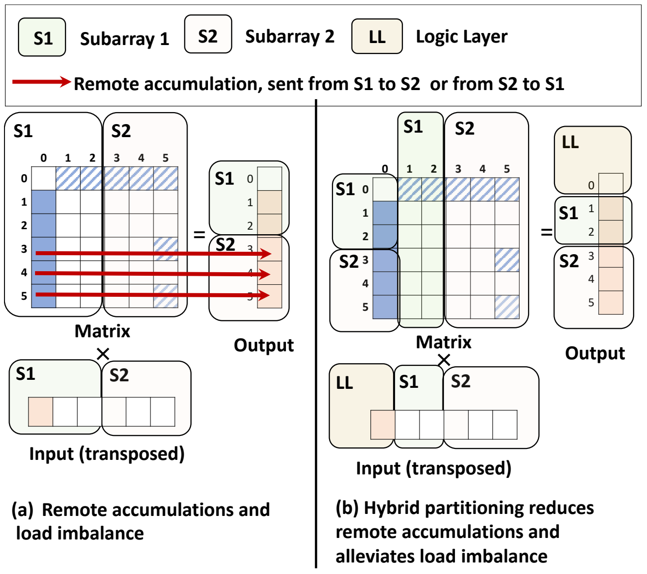

S1 Subarray 1
S2 Subarray 2
LL Logic Layer
Remote accumulation, sent from S1 to S2 or from S2 to S1
Matrix
Output
Input (transposed)
(a) Remote accumulations and load imbalance
(b) Hybrid partitioning reduces remote accumulations and alleviates load imbalance

Figure 2: Remote accumulations and load imbalance. (a) With column-oriented partitioning, long columns cause load imbalance and many remote accumulations. (b) With Hybrid Partitioning, long column entries cause no remote accumulation and no load imbalance.

approach requires processing *every* non-zero element of the input matrix for both SpMV and SpMSpV. On the other hand, for SpMSpV, the column-oriented approach processes only the columns corresponding to the non-zero entries of the input vector. We refer to these columns and their non-zero entries as activated columns and activated entries. As a result, the column-oriented approach is more efficient for SpMSpV [8].

While the column-oriented approach is common in GPU, CPU, FPGA, and ASIC solutions for SpMSpV [11, 26, 27, 38, 41, 42, 49, 53, 54, 59], none of the prior bank-level or subarray-level PIM-based SpMV accelerators [9, 52] have implemented column-oriented processing. To maximize the benefits of column-oriented processing, we need to address two issues: i) random accesses to remote memory segments and ii) *power-law* column length distribution.

Random accesses to remote memory segments:

Processing SpMV and SpMSpV in PIM requires the compressed matrix, the input vector, and the output vector to be partitioned among memory segments. With both row-oriented and column-oriented approaches, the processing units near each segment require access to data that is stored in another memory segment. For example, in Figure 2 (a), one of the multiplication and addition required for generating $Output[3]$ is $Output[3] \mathrel{+}= Matrix[3,0] * Input[0]$. However, Figure 2 (a) shows that $Input[0]$ and $Matrix[3,0]$ reside in Subarray 1 (S1), but $Output[3]$ resides in Subarray 2 (S2). Therefore, the processing unit in S1 does the multiplication part ($Input[0] * Matrix[3,0]$) locally but has to access Subarray 2 (S2) to write the result of multiplication in $Output[3]$.

The remote write accesses are remote accumulations that do not require any mechanism for enforcing the order of operations. Therefore, the result of multiplications can be sent to be accumulated in the destination memory segment. For example, S1 can send the multiplication result to S2 to be added to $Output[3]$ in S2 and continue processing another multiplication and do not need to wait until the accessed operand arrives from a remote memory segment.

Accordingly, we propose *Accumulation dispatching*. In this mechanism, one dedicated subarray in every bank acts as a dispatcher for remote accumulations. Without the dispatcher, each remote accumulation would interrupt the normal processing of the processing unit in the remote subarray. The dispatcher collects all the remote accumulations and sends them to their destination once the destination subarray's processing ends. This solution sacrifices only 6% of capacity. In Section 7.3, we evaluate an alternative albeit impractical approach.

Power-law column length distribution:

Real-world sparse matrices' column lengths follow the *power-law* distribution [19]. That means most of the rows/columns contain very few non-zero entries (referred to as *short* rows/columns), while the remaining row/columns have orders of magnitude higher numbers of non-zero entries (referred to as *long* rows/columns). The natural way of partitioning a matrix for the column-oriented approach is to assign a few full columns to each memory segment, where the input entries that activate these columns reside. However, with a *power-law* column length distribution, this partitioning causes load imbalance, because the processing unit of the subarray that has a long column has to perform many more multiplications than other processing units, whenever this long column gets activated. We also observed that, with naive column-oriented partitioning, most of the remote accumulations are due to long columns.

To address these issues, we propose *Hybrid partitioning* scheme that treats short and long columns differently. We partition the short columns in a normal column-oriented way and store a full short column in one memory segment. However, we distribute the long columns' non-zero entries among all memory segments, so that each non-zero entry and its corresponding entry in the output vector reside in the same memory segment. We also propose hardware support for our proposed partitioning technique. To lower the overhead of our hardware support, we reorder the matrix so that the long columns/rows are the first columns/rows of the matrix and their index is lower than a threshold. As a result, distinguishing the indexes corresponding to these long columns and long rows can be implemented using a comparator and a latch that holds the threshold.

Figure 2 (b) shows that with Hybrid partitioning, the long column no longer causes any remote accumulation, since $Matrix[3:5,0]$ and $Output[3:5]$ reside in the same subarray. This partitioning also alleviates load imbalance, because all processing units co-operate on processing an activated long column.

With Hybrid partitioning, for multiplications, all subarrays need to access the input vector entries that activate long columns. We place these entries in the logic layer (one of the layers in 3D stack memories, introduced in Section 2) and broadcast them to all subarrays. For example, in Figure 2 (b), we place $Input[0]$ in the logic layer.

Based on these two key ideas, we propose *Gearbox*, which adds efficient hardware supports for column-oriented processing to PIM-based accelerators. We use Fulcrum [32] as the baseline PIM architecture for Gearbox. Fulcrum places one lightweight single-word processing unit at every two subarrays to achieve high parallelism. The subarray-level single-word processing allows parallel and independent access per single-word ALU. Therefore, unlike SIMD approaches, the ALUs do not have to wait for all the operands to be collected. However, Fulcrum [32] only supports sequential accesses. It does not support local random accesses (i.e., random access within the same subarray) and remote accesses required by the SpMV and SpMSpV kernels. We modify Fulcrum to add support for a new important range of applications by enabling local random accesses, as well as adding support for our proposed Accumulation dispatching and Hybrid partitioning. Our support for local random accesses, Accumulation dispatching, and Hybrid partitioning is programmable, enabling future works to map more irregular kernels to our proposed architecture.

Our proposed method, Gearbox, with just *one* memory stack, delivers on average (up to) 15.73× (52×) speedup over a server-class GPU, NVIDIA P100, with *three* stacks of HBM2 memory. Note that the P100 is not the state-of-the-art GPU and newer GPUs have even more memory stacks. Compared to GPUs with more memory stacks, Gearbox remains highly competitive in terms of speedup per stack because Gearbox delivers on average 45× speedup per stack compared to NVIDIA P100.

Gearbox also outperforms an ideal model of SpaceA [52], a PIM-based SpMV accelerator that only supports row-oriented processing (assuming no area overhead, perfect load balancing, and no penalty for remote reads for SpaceA) by 58× (447×) per area.

Our paper makes the following contributions:

- Proposing a highly parallel architecture that can exploit the parallelism for regular kernels, as well as SpMV and SpMSpV.
- Proposing the first in-memory-layer approach (near banks/subarrays) that implements column-oriented processing, which is more efficient than row-oriented processing.
- Proposing Hybrid partitioning to reduce remote accumulations and alleviate load balancing.
- Proposing hardware support for remote accumulations and Hybrid partitioning.

2 BACKGROUND

2.1 Memory hierarchy

Figure 3(a) illustrates a 3D stacked memory, where a stack comprises a few memory layers and may include a logic layer. Each memory layer has several banks. Every two or four banks in a layer form a group, and a through-silicon via (TSV) connects the groups in different layers to form a vault.

A bank comprises several subarrays that are connected through a shared global data line (GDL) (Figure 3(b)). To access one column of the data from a bank, a subarray reads an entire row and stores the row in a row-wide buffer, known as the row buffer. Then a column decoder at the edge of each bank selects a column from the row. The selected column traverses the GDL to reach the edge of the bank. Memories vary in row width and column width. We choose memory configurations with short rows (e.g., 2048 bits), such as HMC [20, 32], because memory configurations with short rows are more efficient for parallel row activations and random accesses, where only a few words of a row are useful.

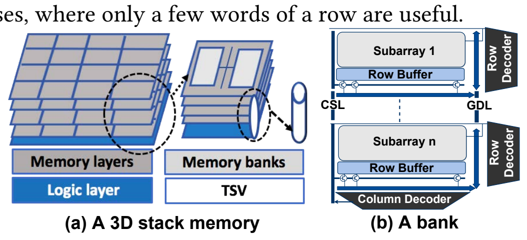


Figure 3: Memory organization: (a) structure of layers and banks, and (b) structure of subarrays.

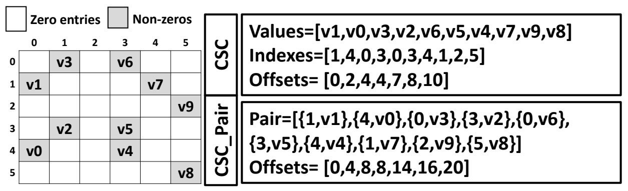
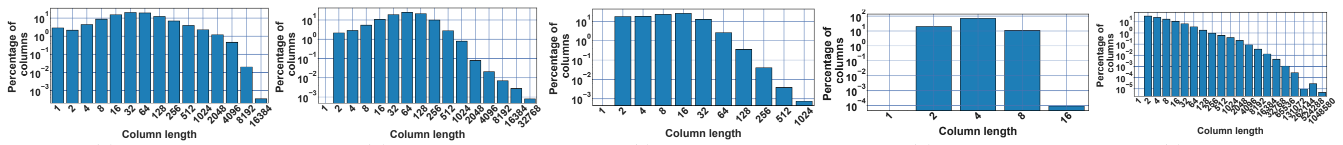

Zero entries
Non-zeros
0 1 2 3 4 5
v3 v6
v1 v7
v9
v2 v5
v0 v4
v8
CSC
Values=[v1,v0,v3,v2,v6,v5,v4,v7,v9,v8]
Indexes=[1,4,0,3,0,3,4,1,2,5]
Offsets= [0,2,4,4,7,8,10]
CSC_Pair
Pair=[{1,v1},{4,v0},{0,v3},{3,v2},{0,v6},{3,v5},{4,v4},{1,v7},{2,v9},{5,v8}]
Offsets= [0,4,8,8,14,16,20]

Figure 4: A sparse matrix in CSC and CSC_Pair format.

2.2 Sparse operations

We denote generalized matrix-vector multiplication as $Output[:] = Matrix[:,:] \times Input[:]$, where $Input[:]$ and $Output[:]$ are vectors, and $Matrix[:,:]$ is a matrix. By "generalized", we mean multiplications and accumulations can be replaced by any other operation with similar properties (e.g., commutativity). In most applications, we need an extra step on the output vector $finalOutput[:] = Output[:] + \alpha y[:]$, where α is a scalar value and $y[:]$ is a vector. The addition and multiplication in this step can also be replaced by any other operation. We refer to this step as Applying.

Many applications can be formulated as SpMV and SpMSpV [9, 41, 42]. For example, Single-Source Shortest Paths (SSSP), a graph processing application, can be formulated as SpMSpV, in which multiplication is replaced by addition, and the accumulation operation is replaced by minimization.



(a) Holly (b) Orkut (c) Patent (d) Road (e) Twitter

Figure 5: Column length distribution in real-world matrices (both x and y-axis are in log scale).

2.3 Sparse matrix representations

There are two main data representations for sparse matrices: (i) compressed sparse rows (CSR) and (ii) compressed sparse columns (CSC). CSC/CSR stores the matrix in three arrays containing: (i) non-zero values ($Values$), (ii) row/column indices of non-zero values ($Indexes$), and (iii) offsets ($Offsets$) that refer to the positions of the start of the columns/rows in both $Values$ and $Indexes$ arrays.

CSC representation is more efficient for column-oriented processing, as it has the position of the start of each column. We can pair the $Values$ and $Indexes$ arrays into one array (CSC_Pair), as shown in Figure 4.

3 MOTIVATION AND KEY IDEAS

3.1 Support for column-oriented processing using accumulation dispatching

Figure 6 shows that the column-oriented algorithm only processes the columns that correspond to non-zero entries of the input vector. Therefore, column-oriented processing operates on the sparse format of the input vector (lines 4-5). We refer to this format of the input vector as the frontier (line 5, currFrontier in Figure 6). Column-oriented processing also requires random access to the output vector (lines 20-21). When we partition the matrix and the input/output vectors among memory segments, the accumulation in line 21 can be *remote* or *local*. For example, in Figure 6, consider a subarray containing $Matrix[:, j : k]$, $Input[j : k]$, and $Output[j : k]$. In line 21, if $j \le row\_index \le k$, the accumulation is a local accumulation. Otherwise, it is a remote accumulation.

```
1   Input:
2       CSC_offsets[0:n]
3       CSC_Pair[0: numNonZeros(Matrix)*2-1],
4       //pair sparse format of the input vector
5       currFrontier[0: numNonZeros(Input)*2-1]
6   Output:
7       OutputDense[0:n-1]=0
8       numOutputNonZeros=0
9       //pair sparse format of the output vector
10      nxtFrontier[0: 2*n-1]
11  //processing only columns corresopnding to non-zeros of the input
12  for (i=0; i< numNonZeros(Input)*2; i+=2):
13      f_column= currFrontier[i]
14      f_value= currFrontier[i+1]
15      col_offset= CSC_offsets[f_column]
16      col_length= CSC_offsets[f_column+1]-CSC_offsets[f_column]
17      for (j=0; j< col_length; j+=2):
18          row_index= CSC_Pair[col_offset +j];
19          row_value= CSC_Pair[col_offset+j+1]
20          //random write to the output vector
21          OutputDense[row_index]+= row_value * f_value
22  //generating the sparse format of the output vector
23  for (i=0; i<n, i++):
24      If(OutputDense[i]!=0):
25          t= numOutputNonZeros*2
26          nxtFrontier[t]= i
27          nxtFrontier[t+1]= OutputDense[i]
28          numOutputNonZeros++
```

Figure 6: The column-oriented algorithm.

Key idea 1: Accordingly, we add hardware support for distinguishing remote accumulations from local accumulations by placing a comparator and two latches that hold the range of index of local accumulations. We also propose a mechanism for dispatching remote accumulations, *Accumulation dispatching*. In this mechanism, one specialized subarray in every bank acts as a dispatcher for the remote accumulations. We elaborate on the details of this mechanism in Section 4.

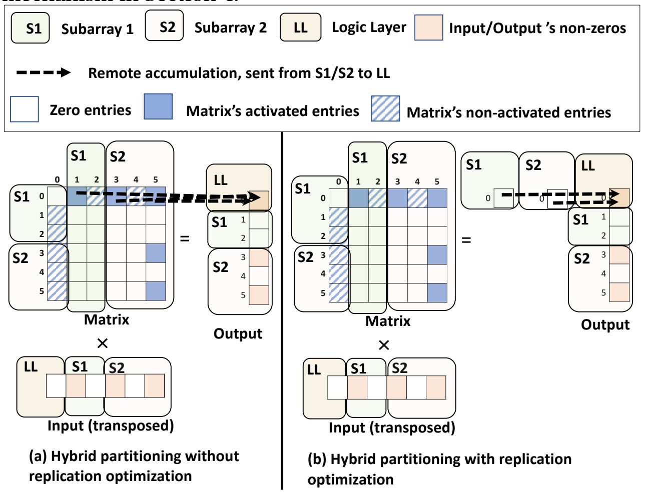

S1 Subarray 1
S2 Subarray 2
LL Logic Layer
Input/Output 's non-zeros
Remote accumulation, sent from S1/S2 to LL
Zero entries
Matrix's activated entries
Matrix's non-activated entries
Matrix
Output
Input (transposed)
(a) Hybrid partitioning without replication optimization
(b) Hybrid partitioning with replication optimization

Figure 7: An extra optimization that replicates the output vector entries corresponding to long rows/columns in each subarray.

3.2 Reducing remote accumulations and balancing the load by supporting Hybrid partitioning

Figure 5 shows the column length distribution of our evaluated datasets, where the x-axis (log scale) shows the column length and the y-axis is the percentage of columns within that range. This figure demonstrates that there are only a few long columns, but they are orders of magnitude longer than the other columns. Same goes for the long rows. We refer to the top $X\%$ (e.g., 0.01%) of columns/rows as long columns/rows. This threshold is configurable in our architecture.

Figure 2 (a) in Section 1 demonstrates that, with column-oriented partitioning, where each subarray has a few full columns, the long columns can cause many remote accumulations and significant load imbalance among processing units.

Key idea 2: Given these observations, to both balance the load and reduce the number of remote accumulations, we propose Hybrid partitioning. Figure 2 (b), in Section 1, illustrates that Hybrid partitioning treats short and long columns differently. We partition

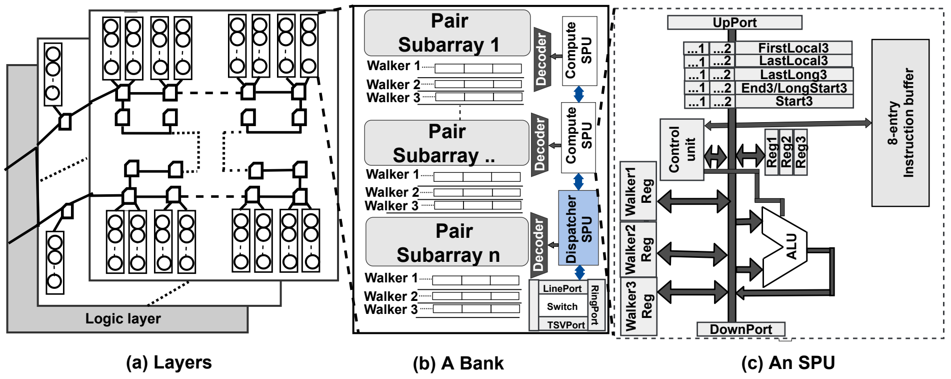


Figure 8: Overall architecture. In (a), the circles are subarrays, the rectangles are banks, and the pentagons are switches.

the short columns in a column-oriented way but divide the long columns among all subarrays. Consequently, each part of a long column resides in the same subarray in which its corresponding part of the output vector resides, eliminating remote accumulations. Furthermore, all subarrays cooperate for processing long columns, alleviating the load imbalance.

In iterative algorithms, the output vector becomes the input vector of the next iteration. Therefore, in the next iteration, all subarrays for multiplication require accessing the output vector entries that activate a long column. We place the output vector entries corresponding to long columns in the logic layer. In the subsequent iterations, we broadcast these entries to all subarrays. Since there are only a few activated long columns in each iteration, the broadcasting imposes negligible overhead. The overhead is evaluated in Section 7.4.

Real-word matrices may also contain a few long rows. Figure 7 (a) shows that these long rows can trigger many remote accumulations. To reduce this remote accumulation overhead, we also place the output entries corresponding to the long rows in the logic layer. The logic layer provides more efficient random accesses, since it has SRAMs.

To implement Hybrid partitioning, our subarray-level processing units should be able to distinguish among input/output entries corresponding to the long columns. We reorder the matrix so that the long columns/rows of the matrix and their index are lower than a threshold. As a result, we can implement this distinction by using a comparator and a latch that keeps the index of the last long column/row. Section 6 explains that this one-time cost is acceptable [17, 19, 52, 57, 60].

To further minimize the overhead of accumulation of long columns/rows, we added an optional optimization, where we replicate the output vectors corresponding to the long columns/rows in all subarrays. Then we accumulate the long rows, first locally in each subarray and then in the logic layer (Figure 7 (b)). If we choose 0.01% of rows/column as long rows/columns, the capacity overhead of this technique stays below 1.7%.

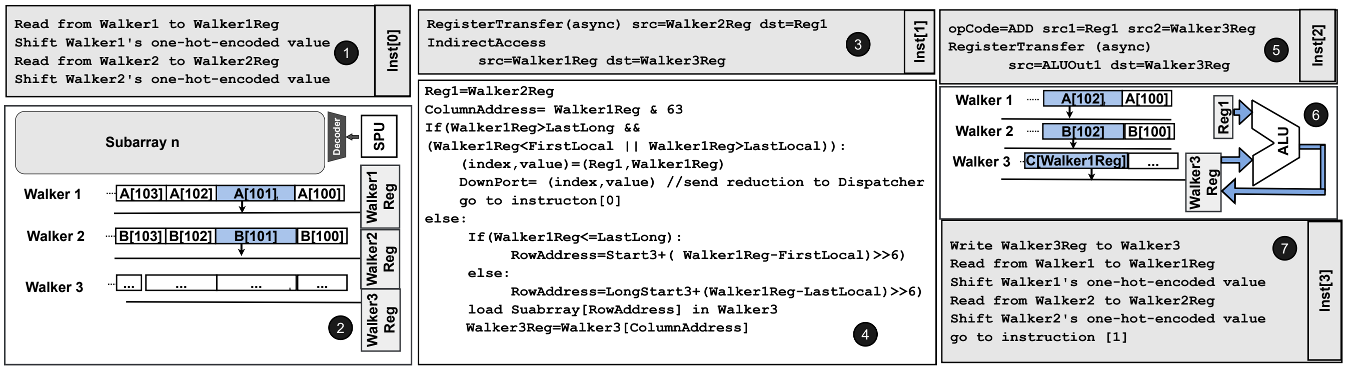
4 PROPOSED ARCHITECTURE

We use Fulcrum[32] as the baseline PIM-based architecture. Motivated by characteristics of memory-intensive applications, where there are few simple operations per loaded datum from memory, Fulcrum places one simplified sequential processing unit per pair of subarrays. Each subarray-level processing unit (SPU) has a few registers, an 8-entry instruction buffer, a controller, and an ALU. In Fulcrum, every pair of subarray has three row-wide buffers, called Walkers. The Walkers load an entire row from the subarray at once, but the processing units sequentially access and process one word at a time. The sequential access is enabled by using a one-hot-encoded value, where the set bit in this value selects the accessed word. Therefore, to sequentially process the row, the processing unit only needs to shift the one-hot encoded value.

We chose Fulcrum because it is more flexible and more efficient than bank-level SIMD approaches for three reasons [34]. First, the three Walkers enable three concurrent sequential accesses. Second, Fulcrum can exploit the parallelism for operations with data dependency because Fulcrum processes row-wide buffers sequentially. Third, Fulcrum can efficiently exploit the parallelism for operations with branches because each subarray has an 8-entry instruction buffer that allows each ALU to perform a different operation independently.

However, Fulcrum can only support sequential accesses and is inefficient for irregular kernels that require random accesses, communications among subarrays, or load balancing. In this paper, we (i) modify the sequential access mechanism of Fulcrum to enable local random accesses, (ii) add in-memory-layer interconnection and a dispatching mechanism to enable remote accumulations, and (iii) add ISA and hardware support for our proposed Hybrid partitioning, which minimizes communications among subarrays and provide hardware support for load balancing. Our modifications add only 10.93% area overhead to Fulcrum but enable exploiting the high parallelism of Fulcrum for a new range of important applications.

Figure 8 illustrates our proposed architecture, which is based on 3D-stacked memories. Similar to prior works [15, 32], every vault has a simple in-order core with a 32KB SRAM scratchpad underneath it, in the logic layer. A ring interconnection topology (Figure 8 (a)) connects the banks in each memory layer. Subarrays within a bank are connected through a line interconnection topology (Figure 8 (b)).



```
Read from Walker1 to Walker1Reg
Shift Walker1's one-hot-encoded value
Read from Walker2 to Walker2Reg
Shift Walker2's one-hot-encoded value
```

```
RegisterTransfer(async) src=Walker2Reg dst=Reg1
IndirectAccess
      src=Walker1Reg dst=Walker3Reg
```

```
Reg1=Walker2Reg
ColumnAddress= Walker1Reg & 63
If(Walker1Reg>LastLong &&
(Walker1Reg<FirstLocal || Walker1Reg>LastLocal)):
    (index,value)=(Reg1,Walker1Reg)
    DownPort= (index,value) //send reduction to Dispatcher
    go to instructon[0]
else:
     If(Walker1Reg<=LastLong):
          RowAddress=Start3+( Walker1Reg-FirstLocal)>>6)
     else:
          RowAddress=LongStart3+(Walker1Reg-LastLocal)>>6)
     load Suabrray[RowAddress] in Walker3
     Walker3Reg=Walker3[ColumnAddress]
```

```
opCode=ADD src1=Reg1 src2=Walker3Reg
RegisterTransfer (async)
       src=ALUOut1 dst=Walker3Reg
```

```
Write Walker3Reg to Walker3
Read from Walker1 to Walker1Reg
Shift Walker1's one-hot-encoded value
Read from Walker2 to Walker2Reg
Shift Walker2's one-hot-encoded value
go to instruction [1]
```

Figure 9: A walk-through example for $C[A[i]]$ += $B[i]$ with four instructions.

As shown in Figure 8 (b), we have two types of SPUs. Subarrays closest to the ring interconnect contain *Dispatcher SPUs*. Other subarrays contain *Compute SPUs*.

The logic layer components launch a kernel (or one step of a kernel) by broadcasting at most 8 instructions to all Compute and Dispatcher SPUs and loading new values from each subarray to the associated latches.

In this section, we elaborate on the role of each part of our architecture, using a simple kernel, $C[A[:]]$ += $B[:]$. At a high level, a Compute SPU reads the i^{th} entry of array $A[:]$, compares this entry against three latches, and processes the accumulation differently based on the result of this comparison. These three latches are $FirstLocal3$, $LastLocal3$, and $LastLong3$. If $FirstLocal3 \leq A[i] \leq LastLocal3$, the accumulation is a local accumulation. If $0 \leq A[i] \leq LastLong3$, the accumulation is again a local accumulation but on the replicated part, $C[0 : LastLong3]$. Otherwise, the accumulation is a remote accumulation. In this case, the Compute SPU sends the index-value pair ($A[i]$ and $B[i]$) to the Dispatcher.

We use this simple example to introduce our modifications to Walkers, provide a walk-through example, and explain the role of Dispatchers. In the end, we elaborate on the details of the instruction format.

4.1 Walkers and indirect accesses

PIM targets memory-intensive applications that process large arrays. In our architecture, each Walker read from or write to one of these large arrays. The Start1/2/3, shown in Figure 8 (c), determine the row address. The End1/2/3 latches determine the end address of the arrays associated with Walker1/2/3, respectively.

For example, Walker1 loads one row from $A[:]$. Then, the controller accesses the row one word at a time by shifting the one-hot-encoded value of Walker1. Once the set bit in the one-hot-encoded value reaches the last position, the controller loads a new row from array $A[:]$.

In our previous example, however, the array $C[:]$ was being accessed randomly using $A[:]$'s entries. When an array is the index of another array, the access is called an indirect access. To enable indirect accesses, we add two fields to the instruction format that determine which register contains the index of the indirect access and which Walker should be used for loading the row containing the accessed word. Our controller derives the row address and column address using the index. To select the accessed word from the row, we shift the one-hot-encoded value and increment a counter until the counter equals the column address. To hide the overhead of shifting, we overlap loading a new row into the Walker and shifting the one-hot-encoded value and use the sub-clock, introduced in [32]. This simple modification enables parallel and independent random access per ALU in the accelerator, enabling applications with high access divergence.

4.2 A walk-through example

Figure 9 illustrates a walk-through example of how our architecture processes the first step of $C[A[i]]+ = B[i]$, using four instructions. The Compute SPUs iterate over these instructions for each entry of $A[:]$ and $B[:]$.

Figure 9 ❶ shows the pseudo format of instruction[0], and Figure 9 ❷ illustrates the operation performed by instruction[0]. With this instruction, the Compute SPUs load one word from Walker1 into $Walker1Reg$ and one word from Walker2 into $Walker2Reg$. With instruction[1], as shown in Figures 9 ❸ and ❹, the SPU moves $Walker2Reg$ to $reg1$ and compares the $Walker1Reg$ against the three latches ($FirstLocal3$, $LastLocal3$, and $LastLong3$). If $FirstLocal3 \leq Walker1Reg \leq LastLocal3$, the Compute SPU derives the row address and column address of $C[Walker2Reg]$, using Start3 latch. If $0 \leq Walker1Reg \leq LastLong3$, the row address is derived using the $LongStart3$ latch that stores the start of the replicated part of $C[:]$ (i.e., $C[0 : LastLong3]$). Using the indirect mechanism that we explained in the previous subsection, SPU loads $C[Walker1Reg]$ into $Walker3Reg$. If the accumulation is a remote accumulation, the controller places the index and the value stored in Reg1 and Walker1Reg on the line interconnection's port (DownPort in Figure 8 (c)) and returns to instruction[0].

Otherwise, Instruction[2], as shown in Figures 9 ❺ and ❻, performs the accumulation ($Walker3Reg$ += $Reg1$).

Instruction[3], as shown in Figure 9 ❼, writes the $Walker3Reg$ register to the Walker3, loads one word from Walker1 into $Walker1Reg$, loads one word from Walker2 into $Walker2Reg$, and returns to Instruction[1]. The controller iterates over these instructions until all A[i] entries are processed.

4.3 Dispatcher and the bank-level switch

The Dispatcher SPUs are located in the subarrays closest to the ring interconnect (Figure 8 (c)). They are primarily responsible for routing remote accumulation packets. To assist in packet forwarding, the Dispatcher SPUs contain a switch that keeps the range of the indexes assigned to its bank and its layer in corresponding latches.

In our previous example, the Compute SPUs send any non-local index-value pairs to the Dispatcher in the bank. When the Dispatcher receives an index-value pair, if the index belongs to its bank, the Dispatcher loads the index-value pair in one of its walkers. If the index-value pair belongs to the same memory layer, the Dispatcher places it on the ring interconnection's port. Otherwise, the Dispatcher forwards the index-value pair to a different memory layer via TSVs. As a result, multiplications and local accumulations are overlapped with sending remote accumulations.

After the multiplication and local accumulation, to complete the remote accumulations, we need two additional steps. In the first step, the Dispatchers start sending the index-value pairs to Compute SPUs in the same bank. In the second step, each Compute SPU processes the received index-value pairs to perform the final accumulation (using instructions that are very similar to the instructions in the first step).

4.4 Maintaining the sparse format of the output vector

In our example, we generated the dense format of the $C[:]$. In iterative algorithms, $C[:]$ may be sparse and the input of another step. A naive way of generating this sparse format is to process $C[:]$ sequentially and generate a list of indexes of non-zero values. Although this step is sequential and highly efficient, when the output vector is very sparse, this step can incur overhead for some applications. We eliminate this step by introducing a few fields in the instruction format. Our controller detects the accumulations that are changing a zero value and acts based on what is programmed by the instruction. To make our support more general, instead of checking only for zeros, we can check for any value. We refer to these values as *clean values*. Different applications may benefit from different clean-values (e.g., all-one value for integers or one of the NaN value representations for floats). We add a latch that keeps this clean-value indicator. Section 5 explains how we use this feature for generating a sparse format of the output vector for SpMSpV.

4.5 Instruction format

Table 1 demonstrates the instruction format of our proposed architecture and lists the bitwidth and description of each field. Our instruction format allows two operations per instruction and concurrent read and write from/to Walkers. The IndirectAccSrc and indirectAccDst fields enable programmable support for indirect access. The LongEntryTreat field adds support for our Hybrid partitioning. CheckCleanVal, CleanValIndxSrc, and CleanPairDst fields enable generating a sparse format of the output vectors.

Table 1: Instruction format of Gearbox

Instruction	Width	Description
NextPC1	3 bits	Program counter of the instructions.
NextPC2	3 bits	
NextPC_Cond	4 bits	Condition that selects between NexpPC1 or NextPC2 as the next instruction.
DecLoop	1 bits	decrement loop counter
OpCode1	4 bits	Opcode of the instructions.
OpCode2	4 bits	
Src1Op1	3 bits	Sources of operation indicated by OpCode1
Src2Op1	3 bits	
Src1Op2	3 bits	Sources of the operation indicated by OpCode2
Src2Op2	3 bits	
ShiftCond1	3 bits	Condition under which the Walker's one-hot-encoded value is shifted.
ShiftCond2	3 bits	
ShiftCond3	3 bits	
ReadWrite1	1 bit	Read from or write to corresponding Walker.
ReadWrite2	1 bit	
ReadWrite3	1 bit	
RegSrc	3 bits	Selects the source and the destination of a register transfer.
RegDst	4 bits	
IndirectAccSrc	2 bits	Register from which the index is read.
IndirectAccDst	2 bits	Walker that loads the row for the indirect access.
LongEntryTreat	1 bit	Determines how to treat long-activating indexes (reduce locally or send downwards).
CheckCleanVal	1 bit	Determines if ALU should check for a clean value
CleanIndexSrc	2 bits	Determines the register containing the index of a clean value.
CleanPairDst	2 bits	Determines whether the clean index should be loaded into a Walker or sent to the Dispatcher.

5 SPMSPV WALK-THROUGH

We can map SpMSpV to our architecture using the following steps.
Step1 (FrontierDistribution): In Section 2, we explained that the sparse format of the input vector is called the frontier. In the first iteration, we partition and distribute the frontier among subarrays. In most algorithms, the first frontier is very small (e.g., one entry for BFS). In iterative applications, the frontier is generated in previous iterations and already resides in subarrays in which their corresponding columns reside, except for the output entries that correspond to long row/columns, which reside in the logic layer. At the start of each iteration, we broadcast the entries residing in the logic layer to all subarrays and append them to the frontier array in each subarray.

```
Input:
    CSC_offsets[0:n]
    //pair sparse format of the input vector
    frontier[0: numNonZeros(Input)*2-1]
Output:
      pack[0: packLength-1]
      packLength
j=0;
// pack frontier value with correponding column information
for (i=0; i< numNonZeros(Input)*2; i=i+2):
    index= frontier[i]
    pack[j]= CSC_offsets[index]
    pack[j+1]= CSC_offsets[index+1]-CSC_offsets[index]
    pack[j+2]= frontier[i+1]
    j=j+3
packLength=j+1
```

Figure 10: OffsetPacking.

Step2 (OffsetPacking): This step packs the column offset, column length, and the values from the frontier array that should be multiplied in the column into a new array. Figure 10 shows the pseudo-code of this step.
Step3 (LocalAccumulations): This step multiplies each value of the frontier with its corresponding column. Figure 11 demonstrates the pseudo-code of this step. In this step, if a clean value is being updated, the clean value indicator and its row index will be sent to the Dispatcher.

```
Input:
      pack[0 : numNonZeros(Input)*3-1]
      CSC_Pair[0: numNonZeros(Matrix)*2-1]
Output:
      OutputDense[0:n-1]
OutputDense[:]=0
for (i=0; i< numNonZeros(Input)*3; i=i+3):
    offset=pack[i]
    length= pack[i+1]
    f_Value= pack[i+2]
    for (j=0; j< length;j=j+2):
        row_index= CSC_Pair[offset+j]
        row_value= CSC_Pair[offset+j+1]
        //the fisrt step for generating nxtFrontier
        if(OutputDense[row_index]==0)
            send (0,row_index) to the dispatcher
        OutputDense[row_index] +=f_Value* row_value
```

Figure 11: LocalAccumulations.

Step4 (Dispatching): In this step, the Dispatcher sends all the stored entries (index-value pairs) to their destination subarrays. Here, the Dispatcher's Walker acts as a buffer.
Step5 (RemoteAccumulations): In this step, the SPU sequentially processes index-value pairs received in the previous step and performs the accumulations. Also, in this step, if the value in the index-value pair is a clean-value indicator, the index of clean-value is appended to the corresponding array.
step6 (Applying): This step processes the array containing the nonzero indexes to generate the frontier for the next iteration, initializes the output vector to clean indicators, and sends long-activating entries to the logic layer to be reduced and applied there. It also performs the apply operation ($finalOutput[:] = Output[:] + \alpha y[:]$, which is explained in Section 2.

6 SOFTWARE STACK

PIM-based accelerators [21, 30, 32, 35] are most efficient for applications that can offload a large dataset to the accelerator once and process any incoming input using the data stored in the accelerator. For example, database tables, as well as matrices for deep learning, graph, and classic machine learning applications, can be offloaded to the accelerator once and used for processing many inputs.

In all these domains, the one-time cost of pre-processing and data placement has typically been considered acceptable. For example, in graph processing, several studies [17, 19, 52, 57, 60] propose pre-processing techniques that improve the execution time. In machine learning applications, the pre-processing time is even more negligible compared to the training cost.
Pre-processing: In Gearbox, we need to partition long columns and replicate the column offset for each partition. To balance the load, we randomize the order of columns assigned to a bank and then reorder the matrix so that the long columns and long rows are the first columns and rows of the matrix.
Data placement: For placing data, we use the offload paradigm. Therefore, an API similar to CUDA's API (cudaMemcpy()) manages the data transfer. We allocate contiguous memory space for each array in each subarray independently and then store the row address of each array as metadata. Then, in each step, we load these metadata in the Start and End latches (as shown in Figure 8 (c) and (d)).
Programming model: Similar to Samsung's PIM [29, 30], we are relying on a library-based programming model, where a compiler would link the kernels in computation graphs of a high-level framework (such as TensorFlow). We will release our assembly library for the evaluated kernels.
Scaling the proposed method for larger datasets: We evaluated our approach using datasets that are as large as datasets evaluated by prior works [7, 52, 57]. Gearbox provides high parallelism in one stack. Therefore, unlike prior works[7, 52, 57], Gearbox does not need multiple stacks for these dataset sizes. However, to extend the architecture for larger datasets, we can use multiple stacks (4-16) per device. To extend the capacity even more, we can connect multiple devices by NVLink3 and NVswitch [5] or similar inter-device interconnection, which allows all-to-all device communications. To extend the algorithm for multiple devices and multiple stacks, we can partition the matrix into several blocks, where each block is assigned to one stack. This technique is used by prior accelerators with limited capacity [9, 22]. In this case, we require an additional step that reduces the results of all blocks. NVLink supports collective operations [3] (e.g., broadcast and allReduce operations) that efficiently support the required inter-device communications for our proposed method. We leave evaluations for multi-device Gearbox as a future work.
Supporting kernels with more than three arrays or more than eight instructions: SpMSpV is an example of a kernel that requires more than three arrays. Since we have only three Walkers, we break the first step of this algorithm into two steps, where each step has three arrays. Given that in-memory-layer PIM-based accelerators with high parallelism target memory-intensive application, with few instructions per loaded data, a few-entry instructions buffer is enough. If a future work identifies a widely-used memory intensive application that require more instruction buffer entries, the instruction buffer can be extended at the cost of higher area overhead. A software solution for mapping a kernel with more than 8 instructions is to break the algorithm into few steps, similar to what we do for SpMSpV.
Handling corner cases: If the amount of remote accumulations is high, the Dispatcher SPU in the LocalAccumulations step or a Compute SPU in the Dispatching step may not find enough space for storing the received index-value pairs. To address this issue, we add a software-hardware-based mechanism. Section 4 explains that each Walker has an *End* latch that indicates the end of its corresponding array. When a Walker reaches the row address that is one less than the row address of the *End* latch, the SPU raises a signal that lets the logic layer know that the reserved space is about to be full. Then the logic layer controller stalls the senders (depending on the step, could be the Compute SPUs or the Dispatchers) and initiates the next step, making the array empty again.

7 EVALUATION

7.1 Methodology

Following prior works [7, 9, 22, 52, 57, 58], we evaluate Gearbox using three graph algorithms and two sparse machine learning kernels: Breadth-First Search (BFS), Page Rank (PR), Single-Source Shortest Path (SSSP), Sparse K-Nearest neighbors (SPKNN), and

Table 2: Configuration details for evaluated architectures

Component	Parameters
GPU	Tesla P100 [1], 12 GB memory 3 HBM2 memory stacks at 549 GB/s (183 GB/s per stack)
Ideal in-logic-layer GPU	512 GB/s per stack [7]
Gearbox	technology:22 nm, 32 vaults 32 subarray, open-bitline structure, 256 bytes per row, 64 banks per layer 8 memory layers, row cycle:50 ns, frequency:164 MHz in-logic-layer components per vault: 4-32 kB SRAM, an ARM Cortex-A35 [15] interconnection: 1.2 GHZ, 64 lane, latency: 0.8 ns for each interconnection segment [20, 52]

Table 3: Evaluated datasets

Matrix	Full name	Rows	Non-Zeros	Density	Size (Bytes)
Holly	hollywood_2009	1139905	112751422	0.0086%	911,130,616
Orkut	soc_orkut	2997166	212698418	0.0023%	1,725,564,672
Patent	cit_Patents	3774768	33037896	0.00023%	294,501,312
Road	road_usa	23947347	57708624	0.00001%	653,247,768
Twitter	soc_twitter-2010	21297772	530051618	0.0001%	4,410,795,120

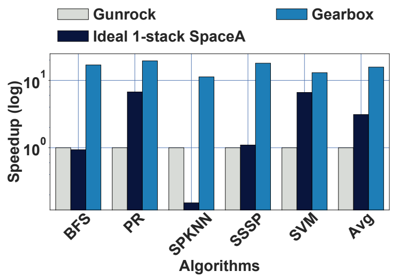


Figure 12: Speedup of our final solution (GearboxV3) against a GPU framework (Gunrock) and a prior work (SpaceA), averaged over datasets. The values less than 10^0 represent slowdown.

Support Vector Machine (SVM). We vary datasets to capture different characteristics of applications for different inputs. Table 3 introduces the datasets, which are real-world matrices from the SuiteSparse matrix collection [14], and Table 2 lists the configurations of the evaluated systems.

There is no established simulator for bank-level and subarray-level computing with simplified processing elements, as these approaches are only recently getting popular. Therefore, we developed an in-house event-accurate simulator for Gearbox and prior works. Furthermore, we integrated our simulator with Gunrock [50] to validate the algorithms. We further evaluate our simulator with assertion testing and analytical evaluations. We developed the RTL model of our SPUs in 14 nm technology and incorporated an overall penalty of 3.08× for processing in 22 nm DRAM. The penalty incorporates the effect of larger technology node and other inefficiencies [28]. Consequently, we evaluated Gearbox with a frequency of 164 MHZ. The frequency of interconnection and one-hot-encoder shifter is 1.2 GHZ.

We evaluated latency, energy consumption, and area of memory elements and interconnect elements using CACTI-3DD [10]. For the breakdown of energy consumption of GPUs, we used Moveprof [4], which is a tool based on integrating NVIDIA's NVProf [6] and GPUWattch [31].

7.2 Speedup

Figure 12 compares our proposed method (GearboxV3) against a server-class GPU and a prior work, SpaceA [52]. Gearbox, with just *one* memory stack, delivers on average (up to) 15.73× (52×) speedup over a server-class GPU, NVIDIA P100, with *three* stacks of HBM2 memory. Gearbox also outperforms an ideal model of SpaceA [52], a PIM-based SpMV accelerator that only supports row-oriented processing. However, SpaceA [52] reports only 4.86% area overhead. Therefore, a fairer comparison is speedup per area. Generously assuming no area overhead, perfect load balancing, and no penalty for remote reads for SpaceA, Gearbox outperforms SpaceA, on average (up to), by 58× (447×) per area. The speedup over SpaceA stems from the fact that Gearbox provides higher parallelism and efficient support for column-oriented processing.

Although we chose Fulcrum as the baseline architecture, the two key ideas can enable column-oriented processing for all PIM approaches. For example, we can add our hardware support for our Hybrid partitioning by adding our latches and comparators to spaceA. Similarly, we can add accumulation dispatching to SpaceA and use the bank-level CAM in SpaceA for accumulating remote results. Our ideas can speed up SpaceA by 3.4 times. Although it is possible to add our ideas to SIMD bank-level architectures (e.g., Newton [24], and Samsung PIM [30]), these architectures require additional modifications, such as in-memory-layer interconnection.

The speedup of Gearbox against GPU stems from three sources: (i) higher internal bandwidth compared to GPU, (ii) lower overhead for random accesses where only a few words out of a cache line is useful, and (iii) inefficiency of SIMD units in GPU for irregular applications.

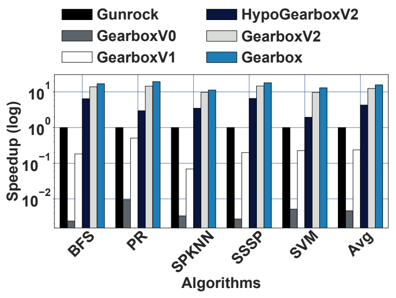


Figure 13: The effect of each optimization. Table 4 lists the description of each Gearbox version.

7.3 The effect of each optimization

Figure 13 illustrates the effect of the proposed optimizations in Gearbox. Table 4 lists the description of each version. GearboxV0 is in fact Fulcrum+local indirect access. Although Fulcrum [32] paper

Table 4: Each Gearbox version shown in Figure 13.

	Description
GearboxV0	row-oriented processing+local random access for accessing a row+ broadcasting the frontier+ using sequential index matching for processing each row
GearboxV1	column-oriented processing+column-oriented partitioning+ our proposed Accumulation dispatching
HypoGearboxV2	column-oriented processing+our Accumulation dispatching+ an impractical partitioning (partitioning the matrix with Hybrid partitioning but placing the entire input and output array in the logic layer)
GearboxV2	column-oriented processing+Accumulation dispatching+ Hybrid partitioning without replication long activating entries in each subarray
GearboxV3	column-oriented processing+reduction dispatching+ Hybrid partitioning+replicating long activating entries

reports speedup for SPMV, the density of the matrix evaluated in Fulcrum is 20%, whereas the density of the evaluated matrix in Gearbox is less than 0.001% (Table 3). Figure 13 shows that, for this density range, GearboxV0 and GearboxV1 are three orders of magnitude and two orders of magnitude slower than Gunrock, respectively. One hypothetical version of Gearbox, HypoGearboxV2, which places the entire input and output array in the logic layer, provides, on average, 4.28× speedup compared to GPU. HypoGearboxV2 is not practical, as SRAMs in the logic layer memory elements do not have enough capacity for the entire input and output vector. GearboxV2, on average, provides 12.48× speedup over GPU by placing only long activating entries of the output/input vectors in the logic layer. The SRAM capacity for this solution is $(2*n\times(4+4)*P/100$, where n is the number of rows and P is the percentage of input/out entries placed in the logic layer. For the evaluated datasets and P of 0.01% , we need 34 KB SRAM in total in the logic layer. GearboxV3 is the final version, whose performance is discussed in Section 7.2.

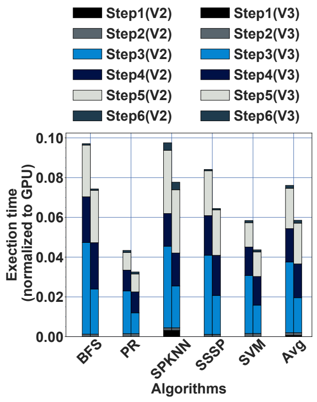
7.4 Execution time and energy breakdown

Figure 14 (a) shows the breakdown of execution time spent on each of the six steps of the algorithm for GearboxV2 and GearboxV3. Here, most of the execution time is spent on LocalAccumulations and RemoteAccumulations. Step1 in this figure includes the overhead of broadcasting of non-zero entries placed in the logic layer, which is on average 1.1% of the total execution time.

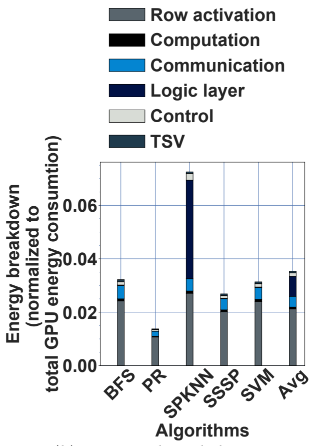
Figure 14 (b) presents the breakdown of the energy consumption of Gearbox, demonstrating that Gearbox reduces the energy consumption, compared to GPU, on average (up to) by 97 (99)%. This figure shows that in most applications, row activations are the major source of energy consumption. The exception is SPKNN, where the input vector and the output vector have many non-zero values corresponding to the long columns/rows, increasing the energy consumption of the operations in the logic layer.

7.5 Comparison against non-in-memory-layer approaches

Figure 15 compares the speedup of Gearbox against three ideal models. The ideal models only account for the overhead of data movement and provide an upper bound for non-in-memory-layer approaches. Figure 15 shows that Gearbox provides 7.94× (31×), on average (up to), speedup per memory stack, compared to the ideal model of a GPU. We also evaluated Gearbox against a purely in-logic-layer approach under aggressive assumptions such as (i) 512 GB/s raw bandwidth, (ii) having enough parallelism to utilize the raw bandwidth, and (iii) having 56 64 kB L1 and 4 MB L2 cache to capture any locality.



(a) Execution time breakdown



(b) Energy breakdown

Figure 14: Breakdown of execution time and energy

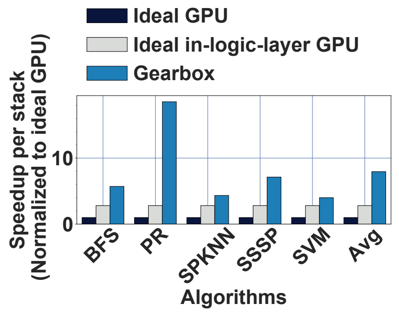


Figure 15: Comparison against ideal models.

Gearbox offers, on average (up to), 2.83× (11×) speedup per memory stack, compared to this ideal model of an in-logic-layer GPU. The main bottleneck of in-logic-layer approaches is the limited bandwidth in the logic layer, which is 29× lower than the bandwidth of in-memory layers. Table 5 compares Gearbox against a few non-in-memory layer approaches based on the reported speedup in their paper on the two common algorithms evaluated by all these accelerators (Page Rank and SSSP). The comparison overestimates the speedup of these accelerators, as we convert their reported CPU speedups to GPU speedups based on the GPU speedups reported in Graphicionado[22], which has no HBM2 memory and has half the memory bandwidth.

Tesseract [7] and GraphP [57] in Table 5 are using HMC-like configuration. Our speedup against these approaches shows that our speedup comes from our in-memory-layer design and not from using HMC-like configuration. Our speedup against these approaches also proves that Gearbox can outperform GPUs with Fine-Grained DRAM [40], with narrow, dedicated TSVs to each bank, similar to HMC.

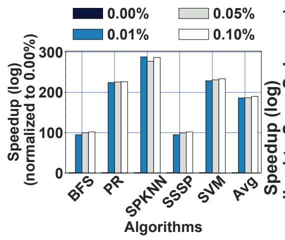
Table 5: Speedup against non-in-memory-layer approaches.

	Graphicionado[22]	Tesseract[7]	GraphP[57]
Per stack/chip	10.01	27.08	21.99
Per area	–	13.47	10.9

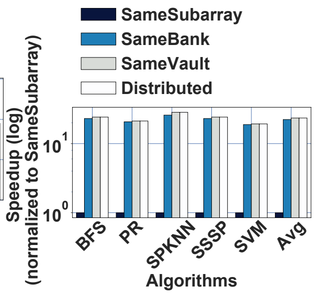
7.6 The effect of load balancing

Figure 16 (a) shows that for most datasets and algorithms, labeling 0.01% of rows/columns as long can significantly improve performance. This figure also shows that increasing the percentage only slightly improves the performance.

We also evaluated the effect of distributing consecutive columns (Figure 16) (b). In real-world matrices, consecutive columns (e.g., neighboring nodes in a graph) are most likely to get activated together. Our evaluations show that distributing consecutive columns among subarrays in a bank (SameBank) provides, on average (up to), 22.3× (76.9×) speedup compared to storing consecutive columns in the same subarray (SameSubarray).



(a) Percentage of long rows/columns



(b) Consecutive columns

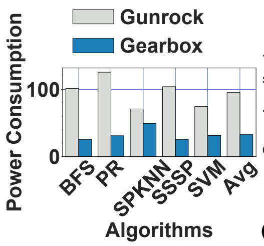
Figure 16: (a) The effect of load balancing techniques.

7.7 Power and temperature constraints

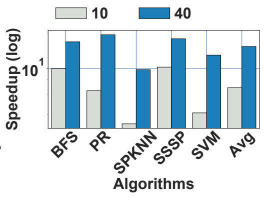
Figure 17 (a) shows that Gearbox reduces power consumption by 75%, compared to the GPU. It consumes, on average, 32.72 watts. Our power density is 465 mW/mm2, which reduces the power density of SpaceA by 12% and is safely under the power density budget of a PIM-based accelerator with a commodity-server active heat sink [16, 56] and under the power budget of the PCIe/CXL peripheral interface. We evaluated the performance of Gearbox under two power budgets: (i) 10W and (ii) 40W. Figure 17 (b) presents the speedup of Gearbox under these two power budgets. To lower the power consumption, we lower the frequency. This figure shows that even under a restricted power budget of 10 watts, Gearbox (with one memory stack) outperforms a high-performance GPU (with three memory stacks), on average (up to) by 6.8× (38.65×).

Table 6: Area evaluation of Gearbox

Component	Area mm^2			
	Per two subarrays		Per layer	
	Optimistic	Pessimistic	Optimistic	Pessimistic
Original DRAM	–	–	–	34.95
Walkers	–	0.011	–	11.26
Bank-level logic and interconnection	–	–	–	4.56
Integer SPUs	0.0067	0.010	6.86	10.42
Float SPUs	0.0098	0.019	10.03	19.45



(a) Power consumption



(b) Performance under power budget

Figure 17: Power and temperature constraints.

7.8 Area evaluation

Table 6 lists the optimistic and pessimistic areas of our hardware components. Our optimistic area numbers are the ones reported by our synthesizer, scaled to 22nm. Our pessimistic area evaluation is the maximum of scaling the optimistic area for 4 layers (using the scale factor derived from [55]) and the pessimistic area reported by our synthesizer. For Walkers, we evaluate the area using CACTI-3DD [10], which is equivalent to pessimistic area evaluations. Gearbox optimistically (pessimistically) imposes 2.42% (10.93)% area overhead compared to a prior work, Fulcrum [32]. In comparison with regular HMC memory, Gearbox optimistically (pessimistically) imposes 73% (100)% area overhead.

7.9 Evaluation for regular kernels

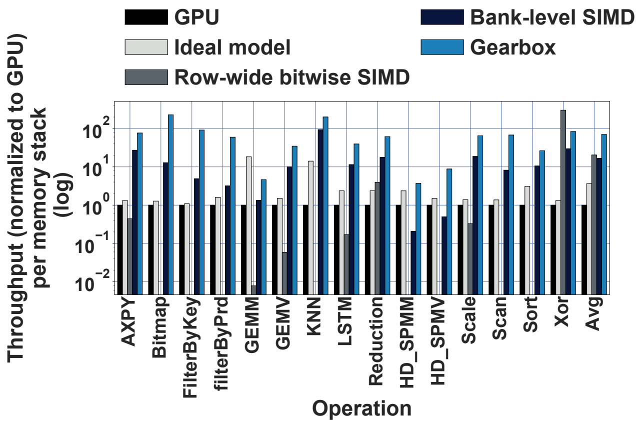


Figure 18: Speedup for regular kernels, reproduced from [34].

GearBox is based on Fulcrum. Therefore, GearBox/Fulcrum can also support and speed up regular workloads. Figure 18 evaluates performance for a range of regular applications from the InSituBench [2] suit. For these evaluations, both Gearbox/Fulcrum and our bank-level SIMD have the same number of ALUs and have the same frequency.

Gearbox provides, on average, 4.4× higher throughput than the bank-level SIMD approach. Gearbox can also outperform DRISA [35], a row-wide bitwise-based SIMD approach, which implements arithmetic operations using bit-wise operations on horizontally laid-out data, by more than two orders of magnitude. SIMDRAM [21], another row-wide bitwise-based SIMD approach that implements arithmetic orations on vertically laid out data, cannot support floating-point operations of the evaluated applications. The vertical layout is also highly inefficient for random accesses, as we would have to activate 32 rows to access a single 32-bit word, one bit per row (the rest of bits in all rows will not be used).

8 RELATED WORK

SIMD and row-wide bitwise approaches: Bank-level SIMD approaches [24, 30] or subarray-level bit-parallel and bit-serial approaches [21, 35, 43–48, 51] perform the same operation on multiple aligned words. These approaches cannot efficiently support SpMV and SpMSpV. Section 7.9 compares Gearbox against these approaches for regular kernels.

Logic-layer-based approaches: This category of prior works [7, 13, 37, 57] employs a few processing units with traditional or decoupled access/execute architectures[25] in the logic layer. These approaches still move data along subarrays, banks, and layers, imposing data movement overheads. Section 7.5 discusses these approaches.

NVM-based techniques: These approaches employ NVM computation capabilities (e.g., CAM capability, analog MAC, and digital computation capabilities) [9, 18, 58]. Due to several issues with NVM-based approaches, including the hardware and energy overhead of analog-to-digital/digital-to-analog converters (which can limit the capacity to 64 MB [9]), low endurance, and high error rate, in this paper, we have focused on DRAM-based accelerators.

Non-PIM approaches: Several ASIC and FPGA designs [12, 22, 39, 41, 42] target SpMSpV and graph processing. The advantage of these approaches is that their performance, similar to other non-PIM approaches, does not highly depend on the data placement in memory. Therefore, they do not require careful offline data placement. For example, they can handle load imbalance at runtime by distributing tasks among processing elements. However, these approaches have to transfer data from memory to the accelerator, imposing data movement overhead. We evaluate Gearbox against Graphicionado [22], an ASIC-based approach, in Table 5. Traditional non-PIM techniques of reducing the cost of data movement, such as prefetching, forwarding, and decoupled access/execute architectures [25, 33] cannot reduce the energy and memory bandwidth and only hide latency. We evaluate Gearbox against an ideal model of these approaches in Section 7.5.

9 CONCLUSIONS AND FUTURE WORKS

Gearbox extends the range of applications that highly parallel PIM-based accelerators can support, by proposing hardware support for Accumulation dispatching, Hybrid partitioning, and subarray-level random accesses.

We can envision three types of future works: (i) extending Gearbox for other irregular kernels, (ii) applying Gearbox in an SRAM/E-DRAM setting, (iii) augmenting Gearbox with a reliability mechanisms for memory technologies with higher error rate. (In this work, we employ Gearbox for graph processing, which is tolerant to error and uses DRAM, where the probability of error per byte in one month, in memory layers, is as low as 1.86375e-8 (85% of DRAM errors caused by the memory controller and memory channel [36])).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback and suggestions. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. We would like to also thank Vaibhave Verma and Akhil Shekar for their valuable comments.

REFERENCES

[1] 2016. Data Sheet: Tesla P100. Retrieved April 22, 2022 from https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf
[2] 2020. A benchmark suit for In-situ computing. Retrieved April 22, 2022 from https://github.com/MarziehLenjani/InSituBench
[3] 2020. Collective Operations. Retrieved April 22, 2022 from https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html
[4] 2020. MoveProf: Integrating NVProf and GPUWattch for Extracting the Energy Cost of Data Movement. Retrieved April 22, 2022 from https://github.com/MarziehLenjani/MoveProf
[5] 2022. NVLink and NVSwitch, The Building Blocks of Advanced Multi-GPU Communication. Retrieved April 22, 2022 from https://www.nvidia.com/en-us/data-center/nvlink/
[6] 2022. Profiler User's Guide. Retrieved April 22, 2022 from https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[7] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *ISCA*.
[8] Ariful Azad and Aydin Buluç. 2017. A Work-efficient Parallel Sparse Matrix-sparse Vector Multiplication Algorithm. In *IPDPS*.
[9] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures. In *ISCA*.
[10] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2012. CACTI-3DD: Architecture-level Modeling for 3D Die-stacked DRAM Main Memory. In *DATE* .
[11] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based Graph Pprocessing Framework on FPGAs. In *FPGA*.
[12] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. Foregraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *FPGA*.
[13] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. GraphH: A Processing-in-memory Architecture for Large-scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.
[14] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.
[15] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. 2017. The Mondrian Data Engine. In *ISCA*.
[16] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. 2014. Thermal Feasibility of Die-stacked Processing in Memory. In *WoNDP*.
[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI*.
[18] Patricia Gonzalez-Guerrero, Tommy Tracy II, Xinfei Guo, Rahul Sreekumar, Marzieh Lenjani, Kevin Skadron, and Mircea R Stan. 2020. Towards on-node Machine Learning for Ultra-low-power Sensors Using Asynchronous $\Sigma\,\Delta$ Streams. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 16, 4 (2020), 1–20.
[19] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A Survey on Graph Processing Accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34, 2 (2019), 339–371.
[20] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. 2017. Demystifying the characteristics of 3dstacked memories: A Case Study for Hybrid Memory Cube. In *IISWC*.
[21] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: a Framework for Bit-serial SIMD Processing using DRAM. In *ASPLOS*.

[22] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics. In *MICRO*.

[23] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*.

[24] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-memory (AIM) Architecture for Machine Learning. In *MICRO*.

[25] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*.

[26] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *ICCAD*.

[27] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical Foundations of the GraphBLAS. In *HPEC*.

[28] Ytong-Bin Kim and Tom W Chen. 1999. Assessing Merged DRAM/logic Technology. *Integration* 27, 2 (1999), 179–194.

[29] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, et al. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *ISSCC*.

[30] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *ISCA*.

[31] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *ISCA*.

[32] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical in-situ Accelerators. In *HPCA*.

[33] Marzieh Lenjani and Mahmoud Reza Hashemi. 2014. Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities. *IET Computers & Digital Techniques* 8, 1 (2014), 30–48.

[34] Marzieh Lenjani and Kevin Skadron. 2021. Supporting Moderate Data Dependency, Position Dependency, and Divergence in PIM-based Accelerators. *IEEE Micro* (2021).

[35] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based Reconfigurable in-situ Accelerator. In *MICRO*.

[36] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. 2015. Revisiting Memory Errors in Large-scale Production Data centers: Analysis and Modeling of New Trends from the Field. In *DSN*.

[37] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *HPCA*.

[38] Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, and Debbie Marr. 2016. Hardware Accelerator for Analytics of Sparse Data. In *DATE*.

[39] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 166–177.

[40] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. 2017. Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems. In *MICRO*.

[41] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An Outer Product based Sparse Matrix Multiplication Accelerator. In *HPCA*.

[42] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SPMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *MICRO*.

[43] Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. 2021. Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration. In *MICRO*.

[44] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. FlexAmata: A Universal and Efficient Adaption of Applications to Spatial Automata Processing Accelerators. In *ASPLOS*.

[45] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *HPCA*.

[46] Elaheh Sadredini, Reza Rahimi, and Kevin Skadron. 2020. Enabling In-SRAM Pattern Processing With Low-Overhead Reporting Architecture. *IEEE Computer Architecture Letters* 19, 2 (2020), 167–170.

[47] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient In-Memory Accelerator for Automata Processing. In *MICRO*.

[48] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. 2017. Ambit: In-memory Accelerator for Bulk Bitwise Operations using Commodity DRAM Technology. In *MICRO*.

[49] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *PPoPP*.

[50] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 1–49.

[51] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. 2021. Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer matching. In *ISCA*.

[52] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *HPCA*.

[53] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.

[54] Carl Yang, Yangzihao Wang, and John D Owens. 2015. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. In *IPDPSW*.

[55] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. In-DRAM Near-data Approximate Acceleration for GPUs. In *PACT*.

[56] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *HPDC*.

[57] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition. In *HPCA*. IEEE, 544–557.

[58] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: Graph Processing in a ReRAM-based Computational Memory.. In *ASP-DAC*.

[59] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. 2019. Hitgraph: High-throughput graph processing framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264.

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. Gridgraph: Large-scale Graph Pprocessing on a Single Machine using 2-level Hierarchical Partitioning. In *ATC*.