SHERLOCK: Scheduling Efficient and Reliable Bulk Bitwise Operations in NVMs

Hamid Farzaneh[1], João Paulo C. de Lima[1,2], Ali Nezhadi Khelejani[3], Asif Ali Khan[1], Mahta Mayahinia[3], Mehdi Tahoori[3] and Jeronimo Castrillon[1,2,4]
first.lastname@{tu-dresden.de,kit.edu}
[1]TU Dresden, [2]ScaDS.AI Dresden/Leipzig, [3]Karlsruhe Institute of Technology, [4]Barkhausen Institut, Germany

ABSTRACT

Bulk bitwise operations are commonplace in application domains such as databases, web search, cryptography, and image processing. The ever-growing volume of data and processing demands of these domains often result in high energy consumption and latency in conventional system architectures, mainly due to data movement between the processing and memory subsystems. Non-volatile memories (NVMs), such as RRAM, PCM and STT-MRAM, facilitate conducting bulk-bitwise logic operations *in-memory* (CIM). Efficient mapping of complex applications to these CIM-capable NVMs is non-trivial and can even lead to slowdowns. This paper presents *Sherlock*, a novel mapping and scheduling method for efficient execution of bulk bitwise operations in NVMs. Sherlock collaboratively optimizes for performance and energy consumption and outperforms the state-of-the-art by 10× and 4.6×, respectively.

KEYWORDS

Compute-in-memory, bulk-bitwise logic, scheduling, reliability

ACM Reference Format:
Hamid Farzaneh[1], João Paulo C. de Lima[1,2], Ali Nezhadi Khelejani[3], Asif Ali Khan[1], Mahta Mayahinia[3], Mehdi Tahoori[3] and Jeronimo Castrillon[1,2,4]. 2024. SHERLOCK: Scheduling Efficient and Reliable Bulk Bitwise Operations in NVMs. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658485

1 INTRODUCTION

Bulk bitwise operations are critical in databases, web search, DNA alignment, encryption, graph processing, networking, and machine learning. Innovative methods like WideTable and BitFunnel harness these operations to accelerate data scans and search engine indexing, as well as many real-world databases support bitmap indices [1]. However, on conventional von Neumann systems, these operations demand substantial data transfer over the memory bus, resulting in high latency and energy consumption.

Recent studies have shown that substantial performance and energy improvements in memory-intensive applications can be achieved by minimizing data movement [2]. One approach for reducing data movement is to exploit the physical characteristics of memory devices, including both conventional SRAM/DRAM and emerging nonvolatile memory (NVM) technologies, and perform logic operations *in-memory* (CIM). This reduces the data movement and allows for unprecedented parallelism in which a bitwise operation on multiple large vectors can be performed in one shot. Numerous recent proposals have leveraged CIM to accelerate memory-intensive applications across various domains either entirely within the main memory [1] or within the storage [3].

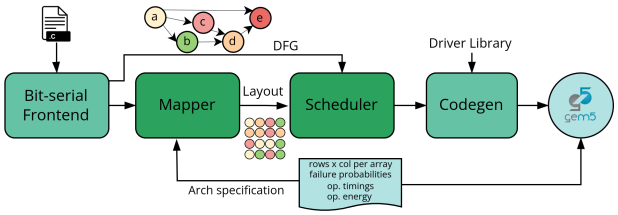


Figure 1: Sherlock compilation flow.

Memristive memories emerge as a promising alternative to traditional DRAM and NAND Flash, offering read latency comparable to DRAM but with smaller cell sizes, denser array layouts, and significantly higher energy efficiency [4]. Memristive devices also support multi-operand bitwise operations, unlike DRAM, which is limited to three inputs [1]. Moreover, memristor-based bit-serial architectures offer unique advantages, with previous studies demonstrating superior energy efficiency and performance compared to their DRAM counterparts [5].

Many existing CIM-logic solutions struggle to handle complex applications wherein different application regions require different logic operations at different granularities. This is primarily due to the fact that efficient mapping of multiple application regions to the memory array, dealing with finer granularities, and managing partial results pose significant challenges and often result in substantial intra and inter-subarray data movement. In the context of NVM-CIM, this becomes even more challenging as it requires careful consideration of write operations and the impact on reliability. Performing a reliable NVM-CIM is challenging. Firstly, the NVM fabrication is not mature enough and susceptible to process variation. Secondly, NVM-CIM has an analog nature, which makes it more sensitive to the impact of process variation. Therefore, there is a probability that the output of the NVM-CIM can be generated incorrectly. This reliability challenge is known as *decision failure*.

Additionally, there is a tradeoff between the performance and decision failure probability of the NVM-CIM. In NVM-CIM realization using the concept of *scouting logic*, the generation of the output is based on the comparison of the resistance from the NVM-CIM array and a reference resistance [6]. Therefore, increasing the number of operands decreases the sense margin and consequently exacerbates the decision failure. However, increasing the number of operands improves the performance. So, balancing performance

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658485

and reliability is a direct tradeoff in NVM-CIM, which is highly technology-dependent [7]. Mapping strategies are hence crucial to effectively exploit these systems and fine-tune them for efficient execution with a proper tradeoff between performance, energy efficiency, and reliability.

To this end, we introduce *Sherlock*, a framework for Scheduling efficient and reliable logical bulk bitwise operations in NVMs, as illustrated in Fig. 1. Sherlock is end-to-end automated, takes a high-level target-agnostic application and a device model as input, and generates optimized code for NVM-CIM accelerators. Internally, Sherlock extracts the data flow graph (DFG) of the input application and implements a novel mapping algorithm optimized for finer-granularities for in-NVM computing. Based on the device model, the scheduler in Sherlock produces optimized code executable on the target CIM system. We evaluate Sherlock on scouting-based CIM systems with different array sizes, different technologies, and different granularities, including the capability to support selective reading and writing. Our results show that Sherlock outperforms the state-of-the-art by 10× and 4.6× in terms of performance and energy consumption while being the first one to consider reliability in terms of decision failure and co-optimize for it.

2 BACKGROUND AND MOTIVATION

This section explains the fundamentals of CIM and outlines our assumptions about the CIM-logic class and overall architecture.

2.1 CIM fundamentals and target system

Memristive devices store data using resistance states: a high resistance state (HRS) representing '1' and a low resistance state (LRS) representing '0'. Similar to conventional memory technologies, NVM devices are organized into arrays of size $m \times n$, i.e., m rows and n columns. All memristive arrays allow for simultaneous activation of multiple wordlines (rows), and the sense amplification process can be adapted to facilitate column-wise logical operations, such as (N)OR, (N)AND, and X(N)OR through a method known as *scouting logic* [6] that operates based on the comparison of the operand's resistance and a reference resistance as shown in Fig .2(a). Additionally, operations like row *copy*, *rotation* of contents in the row buffer, and NOT can be implemented in-place using additional CMOS circuitry in the row buffers. The *copy* operation is usually implemented via row cloning [8]. Nevertheless, the core of our mapping and scheduling solution is independent of the specific hardware approach and applicable to other CIM flavors as well.

The CIM operation(s) implemented by the n columns in a CIM array can be of the same type as in [9] or can be of different types that provide more fine-grained control over operations. The latter is achieved through the utilization of multiplexers connected to both sense amplifiers and reference resistance, driven by configuration bits in the CIM instruction. This column-wise control enables computation on arbitrary data patterns by selectively activating and computing on a few columns of a subarray. It also facilitates the exploitation of instruction parallelism within the same subarray, enabling different operations on distinct sets of columns as long as the operations share the same rows.

2.2 Reliability challenges of CIM in NVMs

Due to their intrinsic process variations, the LRS and HRS values of memristive devices are not fixed values and follow statistical

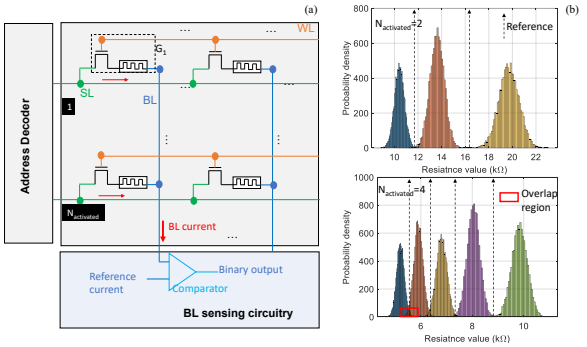


Figure 2: (a) Performing the scouting logic in the NVM crossbar, (b) exacerbation of decision failure during the scouting logic (using STT-MRAM technology), 2 and 4 rows are activated in the top and bottom figures, respectively.

distribution. On top of that, the employed comparator and the reference resistance required by the scouting logic can also be prone to imperfections, leading to the incorrect generation of the NVM-CIM output known as *decision failure*. The probability of decision failure (P_{DF}) can be estimated by the overlap region depicted in Fig.2(b), and is mainly influenced by two factors. First, the **device technology**, as a higher gap between LRS and HRS results in a lower P_{DF}. Compared to STT-MRAM, the PCM and ReRAM devices have a wider gap and hence smaller P_{DF}. Second, the number of **simultaneously activated rows** directly impacts P_{DF}. Increasing the number of activated rows enhances performance but also reduces the sensing margin and significantly increases P_{DF}.

The mapping problem: Most prior works assume that all workloads can use a constant number of rows to buffer intermediate results and all input/intermediate data fit within the rows of a single array. This is unrealistic in NVM technologies as array sizes can not be arbitrarily large [10]. With smaller array sizes, mapping complex applications (larger DFGs) onto CIM arrays is non-trivial. Existing methods traverse the DFG in a breadth-first order and store the operand nodes in the array in the column-major order. This leads to substantial data movement and data duplication. To address this, one must consider the dependencies within the DFG and map operands to a CIM array in a way that prevents operands from one column from depending on operands from other columns, substantially reducing data movement and leading to decreased execution time and energy consumption.

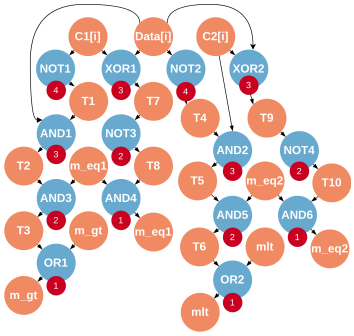
3 SHERLOCK: MAPPING AND SCHEDULING

As illustrated in Fig. 1, Sherlock takes a high-level representation of the input application and generates a DFG. It employs our novel mapping and scheduling algorithm to find an efficient mapping of the operands onto the NVM array and generates efficient code.

3.1 Front-end

For illustration throughout this section, let us consider *bitweaving* as a running example. Fig. 3a shows the pseudo-code for the bitweaving method to evaluate the predicate `BETWEEN C1 AND C2` (column scan in databases). The kernel primarily comprises bulk-bitwise operations, making it well-suited for CIM. The DFG for a single iteration of this application is depicted in Fig. 3b. Note that

```
for (i = 0 to K):
  t1 = NOT(C1[i]);
  T2 = AND(T1, data[i]);
  T3 = AND(T2, meq1);
  m_gt = OR(T3, mgt);
  T4 = NOT(data[i]);
  T5 = AND(T4, C2[i]);
  T6 = AND(T5, m_eq2);
  m_lt= OR(T6, mlt);
  T7 = XOR(data[i], C1[i]);
  T8 = NOT(T7);
  meq1 = AND(meq1, T8);
  T9 = XOR(data[i], C2[i]);
  T10 = NOT(T9);
  meq2 = AND(meq2, T10);
```



(a) Bitweaving code from [11] . **(b) Data flow graph**

Figure 3: Bitweaving implementation of `BETWEEN C1 AND C2` predicate, and its DFG for one loop iteration.

our DFG is always a directed acyclic graph (DAG), where nodes represent operands or intermediate results (orange nodes) and operations (blue nodes), with arrows indicating the dependencies between operations. In this work, we start from a `C` program and employ `pycparser` to produce the abstract syntax tree (AST), which serves as the basis for generating the DAG of the application. All operation nodes are unit-weighted, whereas operand nodes and edges have zero weights. For each operation node in DAG, we compute the b-level [12] (shown in the red circles in Fig. 3b), which represent the priorities of the nodes and are used by the mapping and scheduling algorithm as discussed below.

3.2 Naive mapping algorithm

Algorithm 1 illustrates a naive mapping algorithm to map a DAG to a given hardware target. Unless stated otherwise, our assumptions regarding the hardware target align with those detailed in Sec. 2.1. This algorithm takes the DAG and target specifications as input and produces both a memory layout, indicating how operands (orange nodes) in the application are mapped to the memory array and a generated set of instructions that define the schedule (order) in which operations in the DAG are executed.

Algorithm 1: Naive mapping of a DAG onto a CIM array

```
   Input: target, DAG
   Output: layout, inst
1  /* nq = Queue of op nodes, i = col number, index = curr index of the col   */
2  nq ← [], inst ← [], i ← 1, index ← 1;
3  layout = new layout(target);
4  nq ← b-level-sort(DAG); // sort op nodes by b-level
5  for node in nq do
6      size ← GetUnmappedOpns(node, layout) ; // no. of operands to be map
7      if index + size < m then
8          /* All operands of the op node can fit in this col. n = col size   */
9          layout.update(col_i, node, DAG[node]_{1→size});
10         index ← index + size ;
11     else
12         /* Store as many operands as can fit in the current col.           */
13         layout.update(col_i, node, DAG[node]_{1→(m−index)});
14         /* Move to next (i+1) col., update index based on remain. elements */
15         index ← size - (m-index), i ← i+1;          // m: col size
16         layout.update(col_i, node, DAG[node]_{1→index});
17     end if
18     inst ← inst.append(generate_instructions(node, layout));
19 end for
20 return layout, inst;
```

In this naive algorithm, operands are mapped to the array based on the priorities of their corresponding operations, starting from the first column and proceeding to subsequent columns (from 1 to n, where n is the number of columns in the target array). If an application has a limited number of operands and operations that, along with intermediate results, can fit in a single column of size m, this mapping already delivers the best performance as no data movement is required. However, for real-world applications, a single column is often insufficient, and the naive mapping requires significant data duplication and/or movement.

```
...
op address[arrayID][columns][rows] [cim-op]
write [0][4,8,12,16][932]
Read  [0][1,5,9, 13][5]
Shift [0] R[3]
Read  [0][4,8,12,16][933,934] [XOR,AND,OR,XOR]
...
```

Figure 4: Example of the generated instructions.

In the algorithm, Line 3 initializes the `layout` data structure, which stores data and its arrival cycle in memory. Line 4 computes a list of nodes nq from the DAG, sorted according to the priorities (b-level) of op nodes. The loop from Line 5 to Line 19 iterates over nq for all nodes. For each node, the algorithm first extracts the in-degree and out-degree, representing operands and outputs that need to be mapped to memory, and then checks the layout structure to determine if some operands are already mapped (in case operands are shared by ops). Based on this information, the algorithm computes `size` (Line 6), indicating the number of operands that must be mapped to the array. If the current column ($\texttt{col}_i$) has sufficient free locations to accommodate *size* operands, they are mapped to the memory, and the layout and index values are updated (Lines 9, 10). Otherwise, $m - size$ elements are stored in the current column, and the remaining elements are mapped to the next column (Lines 11-17).

3.2.1 Code generation. After mapping all operands of an op in the DAG, the algorithm proceeds to generate instructions for it (Line 18). Fig.4 presents a snippet of the resulting instructions for an application. The format of these generated instructions is designed to be compatible with our simulation infrastructure. The first field in the instruction indicates the operation type, which can be either read, write, or shift (see Sec.2.1). The second field represents the address, comprising the array ID, columns, and rows. For standard read and write operations, only these two fields are needed. In the case of CIM operations, the third field specifies the logic operation for individual columns. The shift operation is a special operation used for the logical shifting of the row buffer contents for alignment, requiring array ID, shift direction, and shift distance as arguments.

3.3 Optimized mapping

For larger DAGs, the naive mapping leads to substantial data movement. This section explains our novel mapping algorithm, designed to efficiently map DAG operands onto a specified set of columns while taking into account considerations such as data reuse, and effective utilization of the CIM array.

Algorithm 2 presents our mapping approach. It calculates k, the number of columns needed for mapping the operand nodes in the DAG, by dividing the number of operands on the column size (m) (Line 3). Subsequently, it identifies k clusters of op nodes in the DAG, each to be mapped to one of the k columns (Line 5). The size of the clusters (measured as set cardinality $|C|$) is constrained by both $C_{maxSize}$ which is computed from the column size (m) and

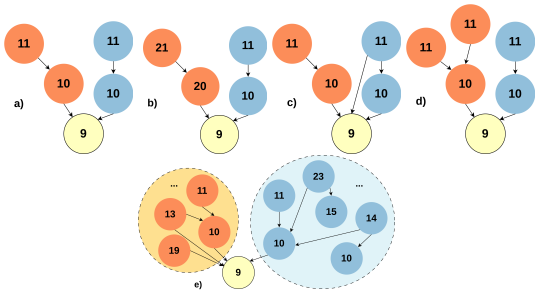


Figure 5: Scenarios of adding a node to a cluster.

the in-/out-degrees of the nodes within the cluster. The algorithm strategically minimizes dependencies between nodes across clusters to reduce data movement between columns.

3.3.1 Clustering DAG op nodes. Similar to the naive mapping, Algorithm 2 also computes the b-level (priority) of the op nodes in the DAG and stores it in *nq* (Line 18). Starting with the highest priority nodes, the algorithm iterates over the node queue. For each node, if it does not have a predecessor, a new cluster is created, and the node is assigned to it (Line 23). In cases where a node already has predecessors, these predecessors have already been assigned to one or more clusters. The assignment of the node to a cluster can follow any of the subsequent scenarios.

Case 1: If there is exactly one predecessor, and the size of the predecessor's cluster is less than $C_{maxSize}$, the node is assigned to this cluster. Otherwise, the node is assigned to a new cluster.

Case 2: If a node has multiple predecessors, and the clusters of the predecessors exhibit similar properties, i.e., same size $|C|$ and identical predecessors priorities (as illustrated in Fig. 5a), the algorithm merges the clusters of the predecessors, provided that the new size does not exceed $C_{maxSize}$. Otherwise, the node is randomly assigned to one of the predecessor's clusters.

If the parent nodes' clusters have different properties, then the algorithm makes the assignment based on the priority difference to the parent node, the dependence of the node on different clusters, and the size of the clusters.

Case 3: When the size and dependencies are identical (see Fig. 5b), the assignment is determined by comparing the smaller priority difference between the current node and its predecessors. In the given example, the priority difference of the node (yellow) with the right parent (blue) cluster is smaller than that with the left cluster, the node is assigned to the right cluster. This is because that node potentially lies in the critical path of the right cluster.

Case 4: In situations like the one depicted in Fig.5c, where the node exhibits greater dependence on one cluster (right in this case), it is assigned to that cluster.

Case 5: In scenarios shown in Fig. 5d, where the node has equal dependency on both clusters and equally influences the critical path, the algorithm assigns it to the cluster with a smaller size to strive at balancing the load across clusters.

In complex situations, such as the one exemplified in Fig.5e, the algorithm calculates an assignment score for all clusters using Eq.1 and assigns the node to the highest scoring cluster. The score of a node d relative to a cluster C is computed as:

$$\text{score}(d, C) = \beta \cdot |C| + \alpha \cdot \sum_{q \in \text{pred}(d) \cap C} \rho(d, q) \qquad (1)$$

Where $\rho(d, q)$ corresponds to the difference of the priorities of nodes d and q, and the constants α and β control the effect of cluster size, dependency, and priority on the assignment.

Eq. 1 covers all the scenarios mentioned above. It maximizes dependent nodes' grouping, as clusters with more connected nodes and lower priority differences yield a higher score. Once a target cluster is determined, the node is added to it (Line 26).

After all clusters in the DAG are computed, and if the number of clusters exceeds k, the algorithm greedily merges them to ultimately have k clusters (Line 30). The merging of clusters follows the same intuition as in the node assignment. The algorithm identifies clusters with maximum dependencies, those that directly influence each other's critical paths, and ensures that the size of the resulting cluster is less than $C_{maxSize}$ before merging them.

3.3.2 Mapping clusters to the CIM array. The k clusters identified in Line 5 are assigned to k columns in the CIM array (Lines 7-14). Since clusters are ensured to be small enough to fit in one column, this assignment is straightforward. However, to maximize parallelism and minimize the number of instructions in the generated code, the algorithm exploits optimization opportunities across clusters by merging operations.

Algorithm 2: Sherlock's optimizing mapping

```
   Input: target, DAG
   Output: layout, inst
1  nq ← [], inst ← [], clusters ← [], i ← 1;
2  layout = new layout(target);
3  k ← operands(DAG) / m;            // No. of cols required for DAG operands
4  /* Find the list of clusters where each cluster is a list of nodes.   */
5  clusters = FindClusters(DAG, k);
6  /* Map Operands of each cluster's nodes to a CIM array column.        */
7  for C ∈ clusters do
8      for node ∈ C do
9          layout.update(col_i, node, DAG[node]);
10         i ← i+1 ;
11         inst ← inst.append(merge(genInst(clusters, layout)));
12     end for
13     i ← 1 ;
14 end for
15 return layout, inst;

16 /* Function to find k op nodes clusters in the DAG.                   */
17 FindClusters DAG, k:
18     pNodes ← [] nq ← b-level(DAG);
19     clusters ← [];     // List of clusters where each cluster is a list of nodes
20     for node in nq do
21         pNodes = pred(node);
22         if pNodes = ∅ then
23             clusters = clusters ∪ {node} ;
24         else
25             tc = GetTargetCluster(node);        // Find target using Eq. 1
26             tc = tc ∪ node
27         end if
28     end for
29     /* Merge smaller clusters to end up having k in total              */
30     clusters ← MergeClusters(clusters, k, C_maxSize);
31     return clusters;
```

3.3.3 Optimizations. If two op nodes, each with x and y operands, respectively, have the same operation type, and the output of one node is solely used once in the other node, a node substitution is performed. This replaces two nodes with a one equivalent node of the same type and $x + y$ operands. This optimization should takes

Table 1: Simulation setup and tools

Technology-level parameters	
STT-MRAM model [14]	- Radius = 20 nm - Barrier Material = MgO - RA = 7.5$\Omega\mu m^2$ - Nominal TMR = 150%
ReReRAM model: *JART VCM v1b Read variability* [15, 16]	- Radius of the filament = 45 nm - Length of the disc region = 0.6 nm - Initial oxygen vacancies concentration: LRS = 3, HRS = 0.009 [$10^{26}/m^3$]
Circuit-level parameters on Cadence virtuoso	
CMOS technology	Global foundry 22FDX
Standard VDD / Temperature	800 mV / 27°C
Array-level parameters on NVSim [13]	
Squared array dim. {Data width (bit)}	128 {512}, 256 {1024}, 512 {2048}, 1024 {4096}
System-level parameters on gem5 [17]	
CPU, ISA, Clock	In-Order CPU, X86, 1GHz
L1I/L1D/L2: size, latency	16/64/256 KiB, 2/2/20 Clock Cycle

into account the characteristics of the target model, such as the simultaneous enabling of rows and their impact on reliability.

Instructions merging across clusters: During instruction generation after the clusters mapping onto the CIM array, our algorithm identifies and merges instructions across clusters. Recall the instruction format from Fig. 4. Suppose the first instruction in cluster1 is `Read [0][1,5,9,13][5]` (from array 0, row 5, read columns 1, 5, 9, and 13), and the first instruction in cluster 2 is `Read [0][6,8][5]` (from array 0, row 5, read columns 6 and 8). The algorithm merges these instructions and issues a single instruction `Read [0][1,5,6,8,9,13][5]`. Before any merging, a dependency check is always performed to ensure that dependencies are not violated. This merging significantly reduces the number of instructions in the generated code.

4 EXPERIMENTAL SETUP AND EVALUATION

We used an extended version of the gem5 simulator for our evaluation. Table 1 summarises our system configuration and parameters. For array-level latency and energy metrics, accounting for hierarchical organization and interconnects, we use NVSim [13]. We conduct SPICE simulations on each NVM cell to determine the resistive levels in LRS and HRS. Then, we use these parameters to calculate the probability of decision failure, represented by the overlap region in Fig.2b), using statistical modeling of the distributions.

We evaluated Sherlock on: *Database:* To utilize the SIMD capabilities of CPUs, BitWeaving [11] proposes two representations: BitWeaving-H and BitWeaving-V. We focus on BitWeaving-V, which we denote as BitWeaving. *Image processing:* Sobel Edge detection is a widely used method that computes the first-order derivative of an image and subsequently determines the disparity in pixel intensities at the edges. We used the bit-sliced implementation proposed in [18]. *Encryption:* AES is a widely used symmetric key encryption algorithm with substitution-permutation operations across multiple rounds to ensure secure encryption. We used the bit-sliced version of AES generated by the Usuba compiler [19].

4.1 Performance and energy comparison

Table 2 illustrates a comparative analysis of applications executed on arrays of varying sizes (512, 1024), utilizing two different technologies (ReRAM and STT-MRAM), and exploiting multi-row activations (MRA) to have operations with exactly 2 operands (slower but reliable) and operations with ≥ 2 operands (faster but less reliable). For MRA of 2, we use the original DAG while for ≥ 2 case, we use the output DAG after applying the transformation explained in Section 3.3.3. The DAG is then mapped onto the hardware via the naive and optimized (opt) algorithms.

Using the optimized method for mapping the Bitweaving kernel on array sizes of 512×512 and 1024×1024 leads to a performance enhancement of 3.49× and 3.1×, for ReRAM and 2.74× and 4.76× for the STT-MRAM, respectively. Having fewer rows in the array requires more columns for executing an application, and achieving an effective mapping is crucial for performance enhancement. In the case of the Sobel application, utilizing the optimized mapping yields a significant improvement of 13.5× and 6.94× in the ReRAM technology over the naive mapping for the 512x512 and 1024 array sizes, respectively. This improvement is because the DAG of the Sobel application is large, which utilizes more columns than the Bitweaving application. AES exhibits similar characteristics as the Sobel and shows similar trends.

In terms of energy consumption, the reduced write and read operations in the optimized mapping lead to an overall average energy improvement of 5.4×. By using the transformation explained in Section 3.3.3, there is an overall reduction in memory operations, directly influencing the naive mapping by consistently lowering latency across all configurations by an average of 1.28×. For *opt*, the MRA ≥ 2 trend is slightly different. The node substitution optimization reduces the overall operations in the DAG but makes it challenging for the mapper to merge instructions. Consequently, for smaller array sizes, there are instances where the energy consumption and latency slightly increase.

4.2 Effect of MRA on application reliability

To examine the impact of operation type and MRA on the application reliability, we compute the probability of at least one failure in an application using the following $P_{app} = 1 - \prod_{i=1}^{N}(1 - P_{DFi})$, where N is the total number of operations and P_{DFi} is the probability of error in operation i. When the application allows for merging boolean operations with >2 operands, the total number of operations is reduced, which could also reduce P_{app}. However, these operations, having higher P_{DF}, contribute to increasing the probability of a single failure in the application.

Fig. 6 shows the effect of increasing the number of operations with more than two operands. The trend shows that when allowing more operations with more than 2 activated rows, which is given by the percentage on top of the data points, latency is improved, with a little sacrifice on reliability. How much the single failure probability reflects on applications' reliability also depends on the memory technology. A $P_{app} < 10^{-4}$ can be considered highly reliable, which is the case for ReRAM (Fig. 6a). In the STT-MRAM, due to the smaller HRS/LRS ration, the operations XOR and OR become much more unreliable. For that reason, we consider the NAND-based implementation of XOR and OR (shown in Fig. 6b). Nevertheless, a $P_{app} \approx 10^{-2}$ with STT-MRAM could be suitable only for applications that tolerate some error.

The optimization technique significantly impacts reliability. On average, *opt* improves P_{app} by 1.5× and 1.3× for ReRAM and STT-MRAM, respectively, while optimizing for execution time and energy consumption In the *naive* optimization, the probability *vs.* latency curve is linear because the choice of which operations to merge does not change with the mapping and scheduling decisions,

Table 2: Energy consumption and latency comparison across memory sizes and optimizations

			naive				*opt*			
		array size ($N \times N$)	1024		512		1024		512	
Tech	Benchmark	# rows in MRA	2	≥ 2	2	≥ 2	2	≥ 2	2	≥ 2
ReRAM	Bitweaving	Latency (μs)	0.36	0.28	1.44	1.13	0.16	0.16	0.41	0.46
		Energy (mJ)	0.18	0.14	0.73	0.57	0.08	0.08	0.21	0.23
	Sobel	Latency (μs)	0.65	0.51	2.61	2.04	0.18	0.19	0.47	0.48
		Energy (mJ)	0.33	0.26	1.32	1.03	0.09	0.09	0.24	0.24
	AES	Latency (μs)	1,147.15	902.29	7111.40	5682.5	29.78	25.32	152.6	155.9
		Energy (μJ)	25.90	20.37	155.32	136.61	2.08	1.77	13.1	10.9
STT-MRAM	Bitweaving	Latency (μs)	0.36	0.28	1.44	1.13	0.16	0.16	0.41	0.46
		Energy (mJ)	0.18	0.14	0.72	0.57	0.08	0.08	0.21	0.23
	Sobel	Latency (μs)	0.65	0.51	2.61	2.04	0.18	0.19	0.47	0.48
		Energy (mJ)	0.33	0.26	1.31	1.02	0.09	0.09	0.24	0.24
	AES	Latency (μs)	124.69	98.07	785.54	637.45	11.25	10.55	73.25	75.57
		Energy (μJ)	4.60	5.40	30.7	23.32	1.45	1.36	8.16	8.7

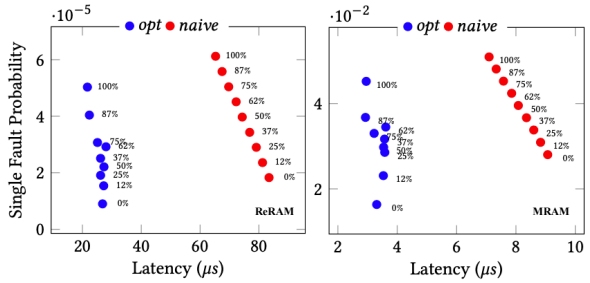


Figure 6: Reliability of Bitweaving output varying the allowed percentage of MRA (> 2 operands)

while in *opt* the choice of the best operations to merge highly depends on these decisions and results in a rather irregular curve. Lastly, from Fig. 6 one can also address the performance-reliability trade-off between ReRAM and STT-MRAM, demonstrating that STT-MRAM is a good memory candidate for accelerating applications that tolerate some deviation in result accuracy.

4.3 Comparison with CPU

Fig. 7 compares the energy-delay-product (EDP) of our optimized configurations compared to the CPU. On average, compared to the CPU, the gains are up to three orders of magnitude. STT-MRAM provides an order magnitude gain in comparison to ReRAM. Bitweaving, sobel, and AES exhibit distinct profiles, showcasing the impact of memory size variations on their energy-delay products.

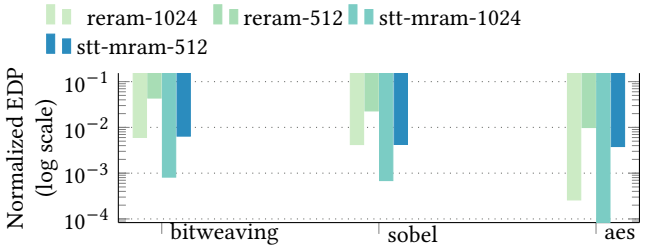


Figure 7: EDP optimizations on performance.

5 CONCLUSION

We present Sherlock, an innovative mapping and scheduling approach tailored for the efficient mapping of bulk bitwise operations in NVMs. For a given DAG and system model, our algorithm intelligently finds operands' mapping onto CIM array columns, with the objective of minimizing data movement and maximizing data reuse and array utilization. We also explore the impact of employing transformations for executing operations with more than two operands on the final reliability and performance. Our evaluation on multiple benchmarks shows that Sherlock significantly improves performance (~10×), energy consumption(~4.6×), and reliability (~1.5×) compared to the state-of-the-art.

ACKNOWLEDGMENTS

This work was partially funded by the Center for Advancing Electronics Dresden (cfaed) and the German Research Council (DFG) through the HetCIM and CIMWARE projects (502388442, 502196634), ROBCOMM project (441857533), and the AI competence center ScaDS.AI Dresden/Leipzig (01IS18026A-D).

REFERENCES

[1] V. Seshadri, D. Lee, T. Mullins, et al. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*, 2017.
[2] A. Boroumand et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *ASPLOS*, 2018.
[3] C. Gao, X. Xin, Y. Lu, et al. Parabit: processing parallel bitwise operations in nand flash memory based ssds. In *MICRO*, 2021.
[4] B. Perach, R. Ronen, B. Kimelfeld, and S. Kvatinsky. Understanding Bulk-Bitwise Processing In-Memory Through Database Analytics. *IEEE Transactions on Emerging Topics in Computing*, 2023.
[5] N. Talati, H. Ha, B. Perach, R. Ronen, and S. Kvatinsky. CONCEPT: A column-oriented memory controller for efficient memory and PIM operations in RRAM. *MICRO*, 2019.
[6] L. Xie, H. Du Nguyen, J. Yu, et al. Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing. In *ISVLSI*, 2017.
[7] M. Mayahinia, C. Münch, and M. B. Tahoori. Analyzing and Mitigating Sensing Failures in Spintronic-based Computing in Memory. In *ITC*, 2021.
[8] V. Seshadri, Y. Kim, C. Fallin, et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *MICRO*, 2013.
[9] S. Li, C. Xu, Q. Zou, et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *DAC*, 2016.
[10] M. S. Truong, E. Chen, D. Su, et al. RACER: Bit-pipelined processing using resistive memory. In *MICRO*, 2021.
[11] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.
[12] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *CSUR*, 1999.
[13] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *TCAD*, 2012.
[14] F. Bernard-Granger, B. Dieny, R. Fascio, and K. Jabeur. SPITT: A magnetic tunnel junction SPICE compact model for STT-MRAM. In *DATE*, 2015.
[15] C. Bengel, A. Siemon, F. Cüppers, et al. Variability-Aware Modeling of Filamentary Oxide-Based Bipolar Resistive Switching Cells Using SPICE Level Compact Models. *TCAS-I*, 2020.
[16] S. Wiefels, C. Bengel, N. Kopperberg, et al. HRS Instability in Oxide-Based Bipolar Resistive Switching Cells. *TED*, 2020.
[17] N. Binkert, B. Beckmann, G. Black, et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 2011.
[18] R. Joshi, M. A. Zaman, and S. Katkoori. Novel Bit-Sliced Near-Memory Computing Based VLSI Architecture for Fast Sobel Edge Detection in IoT Edge Devices. In *iSES*, 2020.
[19] D. Mercadier, P.-E. Dagand, L. Lacassagne, and G. Muller. Usuba: Optimizing & Trustworthy Bitslicing Compiler. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, 2018.