Journal of
Low Power Electronics and Applications

MDPI

Article

PANDA: Processing in Magnetic Random-Access Memory-Accelerated de Bruijn Graph-Based DNA Assembly

Shaahin Angizi [1,*], Naima Ahmed Fahmi [2], Deniz Najafi [1], Wei Zhang [2] and Deliang Fan [3]

[1] Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07103, USA
[2] Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA
[3] Department of Electrical and Computer Engineering, Johns Hopkins University, Baltimore, MD 21218, USA
* Correspondence: shaahin.angizi@njit.edu

Abstract: In this work, we present an efficient Processing in MRAM-Accelerated De Bruijn Graph-based DNA Assembly platform, named PANDA, based on an optimized and hardware-friendly genome assembly algorithm. PANDA is able to assemble large-scale DNA sequence datasets from all-pair overlaps. We first design a PANDA platform that exploits MRAM as computational memory and converts it to a potent processing unit for genome assembly. PANDA can not only execute efficient bulk bit-wise X(N)OR-based comparison/addition operations heavily required for the genome assembly task but also a full set of 2-/3-input logic operations inside the MRAM chip. We then develop a highly parallel and step-by-step hardware-friendly DNA assembly algorithm for PANDA that only requires the developed in-memory logic operations. The platform is then configured with a novel data partitioning and mapping technique that provides local storage and processing to utilize the algorithm level's parallelism fully. The cross-layer simulation results demonstrate that PANDA reduces the run time and power by a factor of 18 and 11, respectively, compared with CPU. Moreover, speed-ups of up to 2.5 to 10× can be obtained over other recent processing in-memory platforms to perform the same task, like STT-MRAM, ReRAM, and DRAM.

Keywords: processing in memory; DNA assembly; SOT-MRAM

Citation: Angizi, S.; Fahmi, N.A.; Najafi, D.; Zhang, W.; Fan, D. PANDA: Processing in Magnetic Random-Access Memory-Accelerated de Bruijn Graph-Based DNA Assembly. *J. Low Power Electron. Appl.* **2024**, *14*, 9. https://doi.org/10.3390/jlpea14010009

Academic Editor: Kenneth S. Stevens

Received: 17 December 2023
Revised: 16 January 2024
Accepted: 31 January 2024
Published: 2 February 2024

1. Introduction

With the advent of high-throughput second-generation parallel sequencing technologies, the process of generating fast and accurate large-scale genomics data has seen significant advancements. Such data can enable us to measure the molecular activities in cells more accurately by analyzing the genomic activities, including mRNA quantification, genetic variant detection, and differential gene expression analysis. Thus, by understanding transcriptomic diversity, we can improve phenotype predictions and provide more accurate disease diagnostics [1]. However, the reconstruction of the full-length transcripts considering sequencing errors is a challenging task in terms of computation and time. Current cDNA sequencing cannot read whole genomes at once [2]. This leads to fragmented data with repeated chunks, duplicated reads, and gaps. Genome assembly aims to merge these fragments into contiguous sequences (i.e., contigs) to reconstruct the original chromosome (Figure 1a) [3].

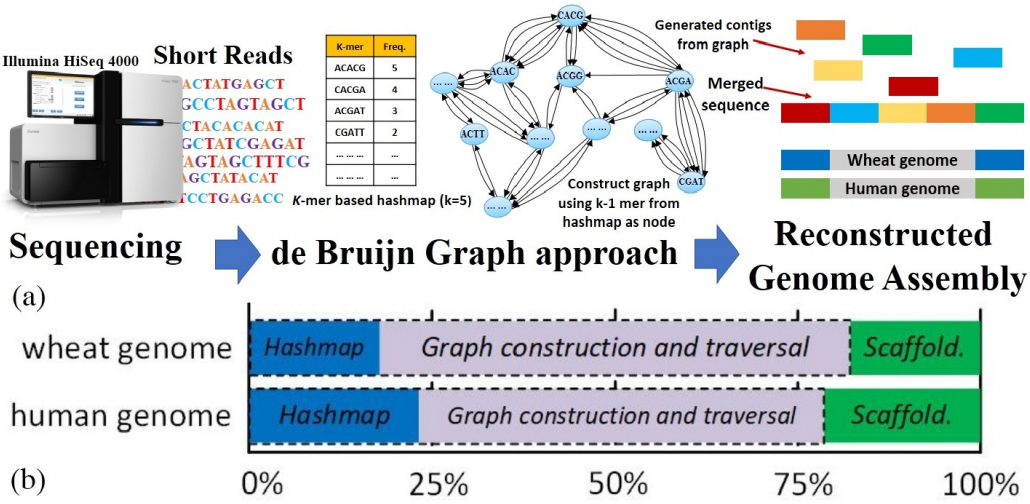


Figure 1. (**a**) The de Bruijn graph-based genome assembly process, (**b**) break down of execution time of Meraculous genome assembler for human and wheat datasets [2,4].

Today's bioinformatic application acceleration solutions are mostly based on the von Neumann architecture, with separate computing and memory components connecting via buses, and, inevitably, consume a large amount of energy in data movement between them [5–8]. The emerging Non-Volatile Memories (NVMs), i.e., Phase-Changing Memory (PCM) [9], Spin-Transfer Torque Magnetic Random-Access Memory (STT-MRAM) [10], and Resistive Random-Access Memory (ReRAM) [11], provide promising features such as high density, ultra-low stand-by power, promising scalability, and non-volatility. In the last two decades, processing in-memory (PIM) architecture, as a potentially viable way to solve the memory wall challenge, has been well explored for different applications [6,12–14]. Meanwhile, processing in non-volatile memory architecture has achieved remarkable success by dramatically reducing data transfer energy and latency [15–18]. The key concept behind PIM is to realize logic computation within memory to process data by leveraging the inherent parallel computing mechanism and exploiting large internal memory bandwidth. Moreover, most of CPU [19,20]-/GPU [20,21]-/FPGA [22]- and even PIM [5,6]-based efforts have only focused on the DNA short read alignment problem, while the de novo genome assembly problem still relies mostly on CPU-based solutions [23]. De novo assemblers fall into Overlap Layout Consensus (OLC), greedy, and de Bruijn graph-based types. De Bruijn graph-based assemblers, gaining attention, solve the problem with the Euler path in polynomial time, unlike the NP-hard Hamiltonian path in OLC-based assemblers [24]. CPU-based assemblers like Velvet [25] and Trinity [26] use the bi-directed de Bruijn graph. GPU–Euler [23,27,28] is among the few GPU-accelerated assemblers. This mainly comes from the nature of the assembly workload that is not only computationally intensive but also extremely data-intensive, requiring very large working memories. Therefore, adapting such a problem to use GPUs with their limited memory capacities has brought many challenges [29]. A graph-based genome assembly process, as shown in Figure 1a, as the main focus of this work, basically consists of multiple stages, i.e., *k*-mer analysis for creating a Hashmap, graph construction, and traversal, and scaffolding and gap closing. Figure 1b depicts a breakdown of execution time for the well-known Meraculous assembler [4] for the human and wheat datasets. We observe that Hashmap and graph construction/traversal are the two most expensive components, which together take over 80% of the total run time.

This motivates us to show that the genome assembly problem and, especially, computationally loaded components can exploit the large internal bandwidth of a Magnetic Random-Access Memory (MRAM) chip for PIM acceleration. Genome assembly heavily relies on *comparison* and *addition* operations, requiring extensive X(N)OR logic computation. However, the intrinsic complexity of X(N)OR logic affects the throughput of PIM platforms [6,15,16,30]. Moreover, in-memory X(N)OR logic involves multi-cycles of majority/AND/OR functions, introducing extra latency and energy consumption through

intermediate data write-back and multi-cycle operations. Intermediate data write-back is necessary as operands for in-memory logic come from the corresponding memory sub-array. In this work, we explore a highly parallel and PIM-friendly implementation of de Bruijn graph-based genome assembly that can accelerate, in particular, the first two stages of the algorithm. Overall this paper makes the following contributions:

(1) We propose a high-throughput comparison/addition-friendly (requiring only one cycle) processing in MRAM architecture for a de Bruijn graph-based genome assembly. We develop PANDA based on a set of innovative microarchitectural and circuit-level schemes to realize a data-parallel computational core for genome assembly;

(2) We reconstruct the existing genome assembly algorithm in a step-by-step fashion to be fully implemented in the proposed PANDA platforms. It supports short read analysis, graph construction, and traversal;

(3) We propose a dense data mapping and partitioning scheme to process the indices locally and handle DNA sequences of various lengths;

(4) We extensively assess and compare PANDA's performance, energy efficiency, and memory bottleneck ratio with a CPU and recent potential PIM platforms.

For clarification, in [30,31], we presented a DRAM-based PIM platform with new ISA to solve three main challenges in the volatile DRAM domain, namely, row initialization, low-throughput X(N)OR logic, and reliability concern about triple row activation. At the application level, we only focused, designed, and discussed the acceleration of DNA assembly's first stage, i.e., Hashmap. We left full algorithm discussion, data partitioning, and mapping implementation of the de Bruijn graph construction (presented as stage two in Section 3.2) and traversal for the Euler path (stage three in Section 3.3) for this submission. The proposed customized and memory-friendly three-stage DNA assembly algorithm in this work could be used for any PIM platform supporting bulk bit-wise X(N)OR and addition operations. The remainder of the paper is designed as follows: Section 2 presents our accelerator design, i.e., the PANDA platform, performance analysis, and software support. Section 3 delineates PANDA's algorithm and mapping support for the genome assembly application. Section 4 is dedicated to our bottom-up evaluation framework and simulation results. Finally, Section 5 concludes this work.

2. PANDA Platform

2.1. SOT-MRAM

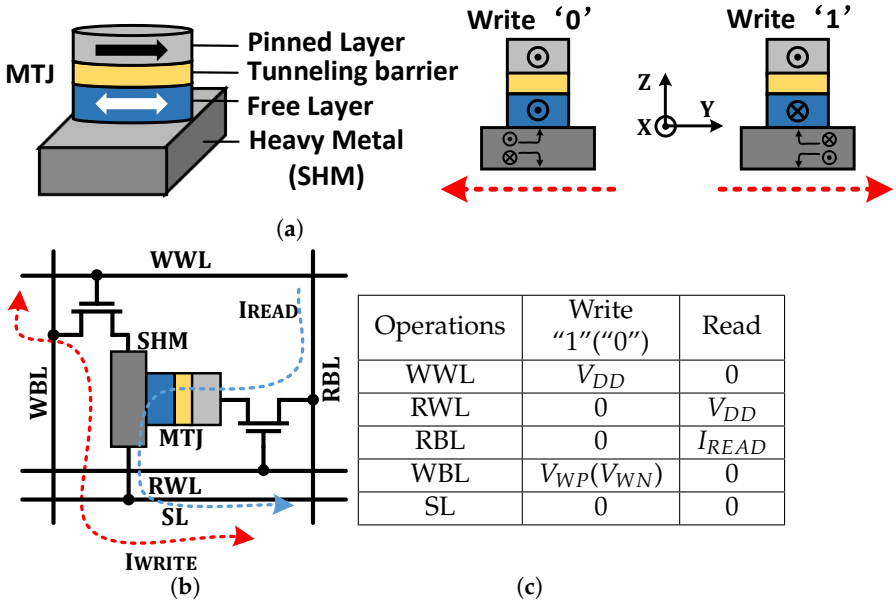
Figure 2a shows the spin–orbit torque magnetic random-access memory (SOT-MRAM) device structure used in this work. The storage element in SOT-MRAM is SHE-MTJ [32], a composite device structure of Spin Hall Metal (SHM) and Magnetic Tunnel Junction (MTJ). The binary data are stored as resistance states of MTJ. Data-"0"(/"1") are encoded as the MTJ's lower (/higher) resistance or parallel (/anti-parallel) magnetization in both magnetic layers (free and fixed layers). Here, the flow of charge current ($\pm$y) through the SHM (Tungsten, $\beta - W$ [33]) will cause the accumulation of oppositely directed spin on both surfaces of the SHM due to spin Hall effect [32]. Thus, a spin current flowing in $\pm$z is generated and further produces spin–orbit torque (SOT) on the adjacent free magnetic layer, causing a switch of magnetization. Each cell located in the computational sub-array is connected with a Write Word Line (WWL), Write Bit Line (WBL), Read Word Line (RWL), Read Bit Line (RBL), and Source Line (SL). The bit-cell structure of 2T1R SOT-MRAM and its biasing conditions are shown in Figures 2b and 2c, respectively.

In this work, the magnetization dynamics of the Free Layer ($\boldsymbol{m}$) are modeled by the LLG equation with spin-transfer torque terms, which can be mathematically described as [32]:

$$\frac{d\boldsymbol{m}}{dt} = -|\gamma|\boldsymbol{m} \times \boldsymbol{H}_{eff} + \alpha(\boldsymbol{m} \times \frac{d\boldsymbol{m}}{dt}) + |\gamma|\beta(\boldsymbol{m} \times \boldsymbol{m_p} \times \boldsymbol{m}) - |\gamma|\beta\epsilon'(\boldsymbol{m} \times \boldsymbol{m_p}) \quad (1)$$

$$\beta = |\frac{\hbar}{2\mu_0 e}|\frac{I_c P}{A_{MTJ} t_{FL} M_s} \quad (2)$$

where $\hbar$ is the reduced plank constant, γ is the gyromagnetic ratio, I_c is the charge current flowing through MTJ, t_{FL} is the thickness of free layer, ϵ' is the second spin transfer torque coefficient, and $\boldsymbol{H_{eff}}$ is the effective magnetic field, P is the effective polarization factor, A_{MTJ} is the cross sectional area of MTJ, and $\boldsymbol{m_p}$ is the unit polarization direction. Note that the ferromagnets in MTJ have In-plane Magnetic Anisotropy (IMA) in the x-axis [32]. With the given thickness (1.2 nm) of the tunneling layer (MgO), the Tunnel Magnetoresistance (TMR) of the MTJ is ∼171.2%.



Operations	Write "1"("0")	Read
WWL	V_{DD}	0
RWL	0	V_{DD}
RBL	0	I_{READ}
WBL	$V_{WP}(V_{WN})$	0
SL	0	0

(c)

Figure 2. (**a**) SOT-MRAM device structure and spin Hall effect, (**b**) schematic, and (**c**) biasing conditions of SOT-MRAM bit-cell.

2.2. Architecture Design

We develop the PANDA platform based on a typical SOT-MRAM hierarchy. Each memory chip consists of multiple memory banks divided into 2D sub-arrays of SOT-MRAM cells, as shown in Figure 3a. We then apply our modification to the sub-array level to make it reconfigurable to support both memory operation and in-memory bit-line computation. As depicted in Figure 3b, the computational memory sub-array (C-Sub.) of PANDA consists of a modified memory row decoder, column decoder, write driver, and reconfigurable Sense Amplifier (SA). The data-parallel intra-sub-array computation of sub-array is timed and controlled using a Controller (ctrl) with respect to the physical address of operands.

PANDA is specially designed to support bulk bit-wise operations between operands stored in each BL. Therefore, the in-memory computational throughput is solely limited by the physical memory row size, i.e., 4 KB/8 KB in modern main memory chips. The storage demands for executing DNA-related processing are consistently substantial. Even when employing von Neumann computing architecture, researchers face the challenge of storing immense data to facilitate any form of processing. The PANDA platform envisions a main memory implementation, ensuring accessibility to a sufficient number of memory cells for in-memory computing. Digital Processing Units (DPUs) are also shared between computational sub-arrays to handle the nonparallel computational load of the platform. In the following, we explain different elements and the supported functions by PANDA.

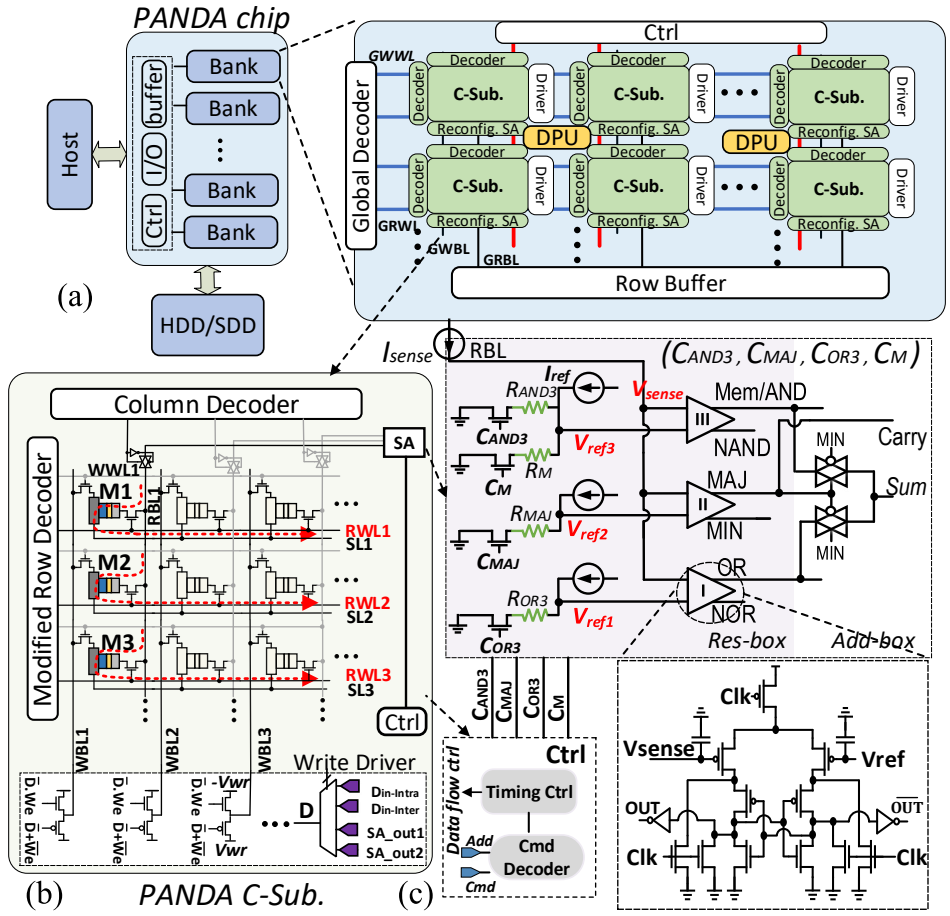


Figure 3. *PANDA* platform: (**a**) memory organization, (**b**) computational sub-array, (**c**) the new reconfigurable sense amplifier designed to implement a full set of 2- and 3-input logic operations.

2.3. PIM Operations

Write Operation: To *write* "0" (/"1") in a cell, e.g., in the cell of the first column and second row (M2 in Figure 3b), the associated write driver first pulls WBL1 to negative (/positive) write voltage. This will provide a preset charge current flow from $-V_{wr}$ to GND ($/+V_{wr}$ to GND) that eventually changes the cell's resistance to Low-R_P/(High-R_{AP}). We considered all the overheads imposed by peripherals to calculate the energy, latency, and area at the application level. This method could be easily replaced with a dual-side single-voltage source write method, where WBL = V_{wr} and SL = 0 writes "1" and WBL = 0 and SL = V_{wr} writes "0" in the memory cell.

Reference Selection and Bit-line Computing: PANDA leverages the reference selection and bit-line computing method on top of a novel reconfigurable SA design, as shown in Figure 3c, to handle memory read and in-memory computation. The main idea of reference selection is to simultaneously compare the resistance state of selected SOT-MRAM cell(s) with one or multiple reference resistors in SA(s) to generate the results. PANDA's SA consists of three sub-SAs with a total of four reference resistors. The ctrl unit could pick the proper reference using enable control bits (C_{AND3}, C_{MAJ}, C_{OR3}, and C_M) to realize the memory read and a full set of 2- and 3-input logic functions, as tabulated in Table 1. We designed and tuned the sense circuit based on StrongARM latch [34], as shown in Figure 3c. Each read/in-memory computing operation requires two clock phases: pre-charge (Clk "high") and sensing (Clk "low"). For instance, to realize the *read* operation, the memory row decoder first activates the corresponding RWL; then, a small sense current (I_{sense}) flows from the selected cell to ground and generates a sense voltage (V_{sense}) at the input of SA-III. This voltage is accordingly compared with the memory mode reference voltage-activated by C_M ($V_{sense,P} < V_{ref,M} < V_{sense,AP}$), as shown in Figure 4a. The SA-III produces high (/low)

voltage if the path resistance is higher (/lower) than R_M (memory reference resistance), i.e., R_{AP} ($/R_P$). PANDA could implement one-threshold in-memory operations ((N)AND, (N)OR, etc.) by activating multiple RWLs simultaneously, and only by activating one SA's enable at a time, e.g., by setting C_{AND3} to "1", 3-input AND/NAND logic can be readily implemented between operands located in the same bit-line. To implement 2-input logics, two rows initialized by "0"/"1" are considered in every sub-array such that functions can be made out of 3-input functions, as indicated by Row Init. in Table 1.

Table 1. Control bits for reconfigurable SA.

Operations	C_{AND3}	C_{MAJ}	C_{OR3}	C_M	Active SA	Row Init. (1)
Read	0	0	0	1	SA-III	No
(N)AND3/(N)AND2	1	0	0	0	SA-III	No/Yes
(N)OR3/(N)OR2	0	0	1	0	SA-I	No/Yes
X(N)OR2	1	1	1	0	SA-I-II-III	Yes
Maj (Carry)/Min	0	1	0	0	SA-II	No
XOR3 (Sum)	1	1	1	0	SA-I-II-III	No

(1) Row initialization is needed to convert a three-input PIM operation to a two-input operation. For instance, XOR2 function is made of XOR3 output by setting one of the memory operands to binary "0".

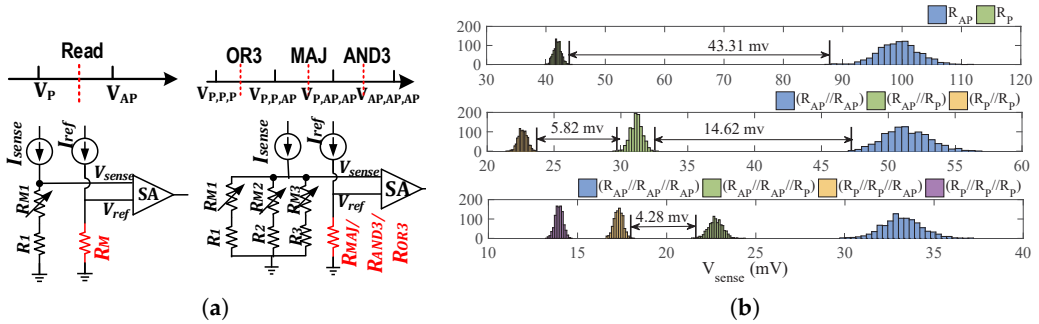


Figure 4. (**a**) Reference comparison to realize in-memory operations, (**b**) Monte Carlo simulation of V_{sense}.

Addition: PANDA's SA is enhanced with a unique circuit design that allows single-cycle implementation of addition/subtraction (add/sub) operation quite efficiently. By activating three memory rows at the same time (RWL1, RWL2, and RWL3 in Figure 3b), OR3, Majority (MAJ), and AND3 functions can be readily realized through SA-I, SA-II, and SA-III, respectively. Each SA compares the equivalent resistance of parallelly connected input cells and their cascaded access transistors with a programmable reference by SA ($R_{OR3}/R_{MAJ}/R_{AND3}$). The idea of voltage comparison between V_{sense} and V_{ref} to realize these functions is depicted in Figure 4a. While there are several addition-in-memory designs in the non-volatile memory domain, they typically apply a large circuitry after SA to realize a multi-cycle design. In order to implement a single-cycle addition operation, we then reformulate the full-adder Boolean expression to make it PIM-friendly. We noticed when the majority function of three inputs is 0, the Sum can be implemented by OR3 function, and when the majority function is 1, Sum can be achieved through AND3 function. This behavior can be implemented by a multiplexer circuit shown in Add-box in Figure 3c. The Boolean logic of such in-memory addition function is written as:

$$Carry = AB + AC + BC = Maj(A, B, C) \quad (3)$$

$$\begin{aligned} Sum &= ((\overline{AB + AC + BC}) \cdot (A + B + C)) + ((AB + AC + BC) \cdot ((ABC)) \\ &= \overline{Maj(A, B, C)} \cdot (OR(A, B, C) + MAJ(A, B, C) \cdot (AND(A, B, C) \\ &= \overline{Carry} \cdot (OR(A, B, C) + Carry \cdot (AND(A, B, C) \end{aligned} \quad (4)$$

The carry-out of the full-adder can be directly produced by the MAJ function (Carry in Figure 3c) just by setting C_{MAJ} to "1" in a single memory cycle. For the MAJ operation, R_{MAJ} is set at the midpoint of $R_P//R_P//R_{AP}$ ("0","0","1") and $R_P//R_{AP}//R_{AP}$ ("0","1","1"), as depicted in Figure 4a. Here, assuming the M1, M2, and M3 operands (Figure 3b), the PANDA can generate Carry-MAJ and Sum-XOR3 in-memory logics in a single memory cycle. The ctrl's configuration for such an add operation is tabulated in Table 1.

It is noteworthy that the PANDA architecture does not rely on a specific NVM technology or cell structure; it operates effectively as long as the technology is resistive-cell-based, such as PCM and RRAM. Our experiments show that utilizing PCM and RRAM cells with a high ON/OFF ratio in the PANDA architecture results in a significantly larger read margin compared to SOT-MRAM. This, in turn, leads to higher reliability even when activating more rows (e.g., up to a 64-row operation for PCM), while it is possible to use other emerging NVMs for better read margin, it is worth noting that PCM and RRAM cells generate a larger sensing voltage, leading to higher power consumption compared to STT-MRAM and SOT-MRAM. In summary, PANDA based on SOT-MRAM sacrifices some sense margin for lower power consumption. Other NVMs may provide a larger sense margin but at the cost of energy efficiency.

Comparison: The PANDA platform offers a single-cycle implementation of XOR3 in-memory logic (Sum). To realize the bulk bit-wise comparison operation based on XNOR2, one memory row in each PANDA's sub-array is initialized to "1". In this way, XNOR2 can be readily implemented out of the XOR3 function. Therefore, every memory sub-array can potentially perform parallel comparison operations without the need for external add-on logic or multi-cycle operation.

2.4. Performance Analysis

Functionality: To verify the circuit functionality of PANDA's sub-array, we first model the SOT-MRAM cell by jointly applying the Non-Equilibrium Green's Function (NEGF) and Landau–Lifshitz–Gilbert (LLG) with spin Hall effect equations [6,32]. We then develop a Verilog-A model of a 2-transistor–1-resistor SOT-MRAM device, with the parameters listed in Table 2 to co-simulate with the other peripheral CMOS circuits displayed in Figure 3 in Cadence Spectre and SPICE. We use the 45 nm North Carolina State University (NCSU) Product Development Kit (PDK) library for our circuit analysis. The transient simulation result of a single 256 × 256 sub-array is shown in Figure 5. We take M1, M2, and M3 as three SOT-MRAM cells located in the first column as the inputs for our evaluation. Here, we consider four input combination scenarios for the write operation, as indicated by 000, 100, 110, and 111 in Figure 5. For the sake of clarity of waveforms, we assume a 3ns period clock synchronizes the write and read operation. However, a 2ns period can be used for a reliable read and in-memory computation.

During the precharge phase of SA (Clk = 1), $\pm V_{write}$ voltage is applied to the WBL to change the MRAM cell resistance to R_{low} = 5.6 kΩ or R_{high} = 15.17 kΩ. Prior to the evaluation phase (Eval.) of SA, WWL and WBL are grounded, while RBL is fed by the very small sense current, I_{sense} = 3 μA. In the evaluation phase, RWL goes high and, depending on the resistance state of parallel bit-cells and, accordingly, SL, V_{sense} is generated at the first input of SAs when V_{ref} is generated at the second input of SAs. A voltage comparison between V_{sense} and V_{ref} for AND3 and OR3 and the output of SAs are plotted in Figure 5. For example, we observe only when $V_{sense} > V_{ref,AND}$ (M1M2M3 = 111), the SA-III outputs binary "1", whereas the output is "0" . Figure 5 also shows the in-memory XOR3 function (Sum) accomplished in a single memory cycle through three SA outputs.

Table 2. Device parameters.

Parameter	Value
Free layer dimension $(W \times L \times t)_{FL}$	$60 \times 40 \times 2$ nm^3
SHM dimension	$60 \times 80 \times 2$ nm^3
Demagnetization factor, D_x; D_y; D_z	0.066; 0.911; 0.022
Spin flip length, λ_{sh}	1.4 nm
Spin Hall angle, θ_{sh}	0.3
Gilbert damping factor, α	0.007
Saturation magnetization, M_s	850 kA/m
Oxide thickness, t_{ox}	1.2 nm
RA product, RA_p/TMR	10.58 $\Omega{\cdot}\mu$m^2/171.2%
Supply voltage	1 V
CMOS technology	45 nm
SOT-MRAM cell area	69 F^2
Access transistor width	4.5 F
Cell aspect ratio	1.91

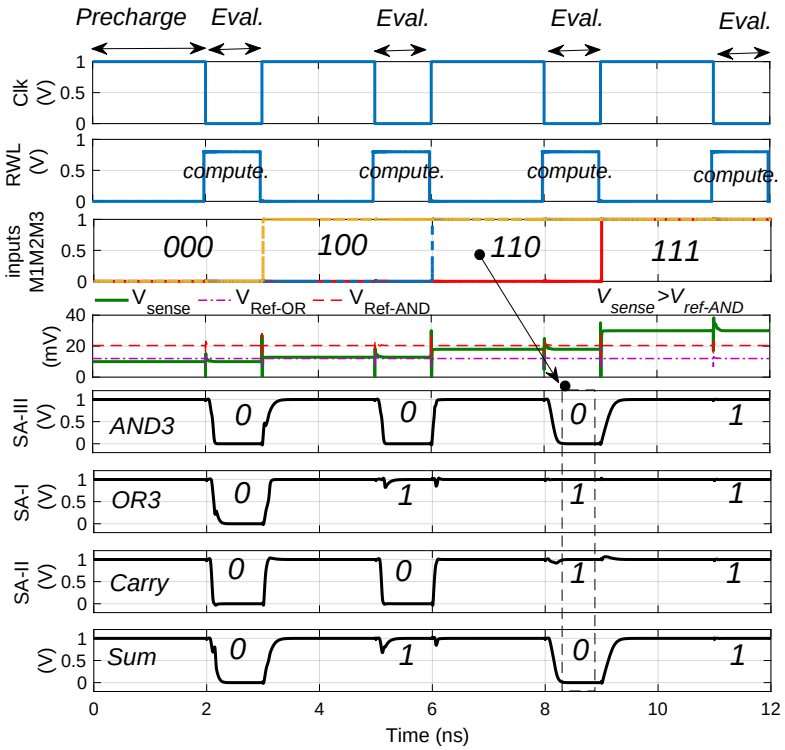


Figure 5. Transient simulation wave-forms of PANDA's sub-array and its reconfigurable SA for performing single-cycle in-memory operations.

Reliability: We assess the variation tolerance in the proposed sub-array and SA circuit by running a Monte Carlo simulation. We run the simulation for 10,000 iterations considering two sources of variations in SOT-MRAM cells: first, a $\sigma = 5\%$ process variation on the Tunneling Magnetoresistance (TMR) and, second, a $\sigma = 2\%$ variation on the Resistance–Area product (RA_P). The results illustrated in Figure 4b prove that the sense margin reduces by increasing the number of selected input cells for in-memory operations. We observe the sense margin for three-input in-memory logic is relatively small, wherein the P//P//P and P//P//AP margin shows a minimal ∼3mV margin. Such a sense margin could be enhanced by either increasing oxide thickness (t_{ox}) or the sense current, but obviously by imposing more power consumption. To show this, we first explore such a worst-case scenario in a 3-input logic voltage margin considering the different stochastic variations on MTJ's RA_P/TMR (2%/5%, 5%/2%, and 5%/5%) in Figure 6a by increasing t_{ox}, from 1 nm to 3 nm, as experimentally demonstrated in [35]. We correspondingly

plot the TMR values achieved from our experimentally benchmarked model in the top x-axis. Note that I_{sense} is set to 5 µA. We observe increasing t_{ox} from 1.2 nm to 2.6 nm (corresponding to 171.2% to 211% increase in TMR value) increases the sense margin by ∼28.4 mV, which considerably improves the reliability of the OR3 operation in PANDA. Such an increase in TMR will enlarge the margin for MAJ and AND3 operations as well. Furthermore, we performed an extensive circuit-level Monte Carlo statistical analysis to investigate the process variation effects on the triple row activation mechanism considering ±10% variation and noises in different components of the MRAM array, such as RBL capacitance, SOT-MRAM bit-cell's access transistor, and SA (width/length of transistors). Figure 6b shows the experiment results of the sense voltage margin for memory read, and all supported 2-/3-input in-memory operations with a Gaussian-distributed variation (3σ) added to the parameters. We gradually increase the I_{sense} to plot the impact of sense current in the process. We observe that the larger I_{sense} is, the larger the voltage margin achieved for different operations. Based on this, a ∼15 µA increase in sense current will lead to a ∼25 mV increase in the OR3 case.

PANDA architecture does not necessarily rely on a certain NVM technology or cell structure. As long as the technology is based on resistive cells, i.e., Phase-Changing Memory (PCM) and Resistive Random-Access Memory (ReRAM), PANDA can readily perform in-memory computation. Based on our experiments, leveraging PCM and ReRAM cells (with a high ON/OFF ratio) in PANDA architecture leads to a significantly larger read margin compared with SOT-MRAM, which further translates to higher reliability even by activating more number of rows (e.g., up to 64-row operation for PCM). Therefore, it is possible to use other types of emerging NVMs to achieve a better read margin. Notwithstanding, PCM and ReRAM generate a larger sensing voltage and consequently consume more power compared with SOT-MRAM. In conclusion, sacrificing the sense margin provides PANDA (based on SOT-MRAM) with a lower power consumption. However, a larger sense margin can be obtained using other NVMs forfeiting energy efficiency.

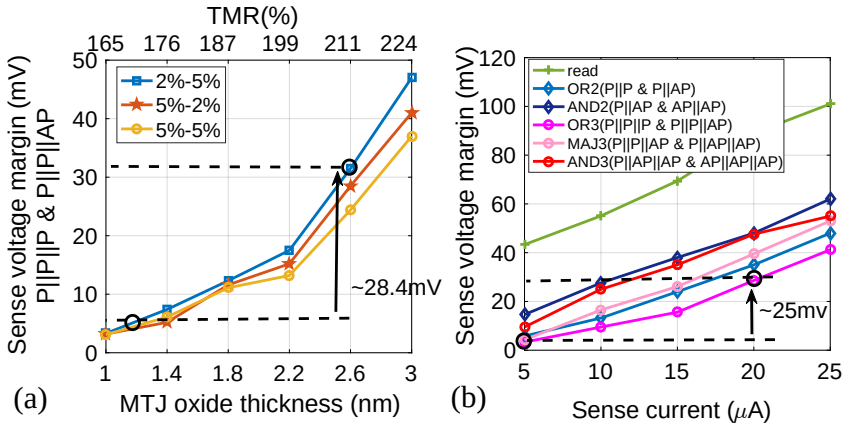


Figure 6. (**a**) Sense voltage margin of 3-input operation between P//P//P and P//P//AP cases vs. t_{ox} and TMR with different variations in RA_P/TMR with a fixed $I_{sense} = 5$ µA. (**b**) Voltage margin between sensitive states of PANDA's operations vs. I_{sense} with a t_{ox} = 1.2 nm and 10% variation added to the array.

Sub-array level Performance: To explore the hardware overhead of PANDA on top of a standard unmodified SOT-MRAM platform, we perform an iso-capacity performance comparison. We develop both platforms with a sample 32 Mb single bank, 512-bit data width in the NVSim memory evaluation tool. The circuit-level data are adopted from our circuit-level simulation and then fed into an NVSim-compatible PIM library to report the results. Table 3 lists the performance measures for dynamic energy, latency, leakage power, and area. We observe that there is a ∼30% increase in the area to support the proposed in-memory computing functions for genome assembly. As for dynamic energy, PANDA shows an increase in R (Read) energy despite the power gating mechanism used in the

reconfigurable SA to turn off non-selected SAs (SA-I and -II while reading operation). In this way, C-Add (C stands for Computation) requires ∼2.4× more power compared with a single SA read operation. However, Table 3 shows PANDA is able to offer a close-to-read latency for C-AND3 and C-Add compared with the standard design. There is also an increase in leakage power obviously coming from the add-on CMOS circuitry.

Table 3. Performance comparison between an standard SOT-MRAM chip and PANDA.

Designs	Area (mm^2)	Dynamic Energy (nJ)				Latency (ns)				Leak. Power (mW)
		R	W	C-AND3	C-Add	R	W	C-AND3	C-Add	
Standard	7.06	0.57	0.66	-	-	3.85	4.5	-	-	402
PANDA	9.3	0.78	0.69	0.85	1.93	3.91	4.59	3.91	3.91	586

2.5. Software Support

PANDA is designed to be an efficient and independent accelerator for DNA assembly; nevertheless, it needs to be exposed to programmers and system-level libraries to use it. PANDA could be directly connected to the memory bus or through PCI Express lanes as a third-party accelerator. Thus, it could be integrated similar to that of GPUs. Therefore, an ISA and a virtual machine for parallel and general-purpose thread execution need to be developed like NVIDIA's PTX. With that, at install time, the programs are translated to the PANDA's ISA discussed here to implement the in-memory functions listed in Table 1. We introduce *PANDA_Mem_insert* (des, src, size) instruction to read source data from the memory and write it back to a destination memory location consecutively. The size of input vectors for in-memory computation could be at most a multiple of PANDA's sub-array row size. *PANDA_Cmp* (src1, src2, size) performs parallel bulk bit-wise comparison operation between source vector 1 and 2. *PANDA_Add* (src1, src2, size) runs element-wise addition between cells located in a same column as will be explained in next section.

Regarding software reliability, most Error Correcting Codes (ECC)-enabled DIMMs rely on calculating some hamming code at the memory controller and use it to correct any soft errors. Unfortunately, such a feature is not available for most PIM platforms [36] including PANDA, as the data being processed are not visible to the memory controller. Employing in-memory error-correcting code techniques [37] is vital in future PIMs to maintain data reliability in the presence of computation mechanisms using memory and growing noise and reliability problems. To overcome this issue, PANDA can potentially augment each row with additional ECC bits that can be calculated and verified at the memory module level or bank level. Augmenting PANDA with reliability guarantees is left as future work.

3. PANDA Algorithm and Mapping

The genome assembly algorithm consists of three main stages, as visualized in Figure 7. First, breaking down each short read in the sequence into smaller chunks of *k*-mers in a consecutive manner and keeping the frequency of each distinct *k*-mer in a Hashmap; second, generating a de Bruijn Graph out of the Hashmap; third, traversing through the de Bruijn Graph to reconstruct the entire genome using the Euler Path traversal concept (stages II and III are so-called contig. generation). There is a final stage called scaffolding to close the gaps between contigs, which is the result of the de novo assembly [2].

The first three stages always take the largest fraction of execution time and computational resources (over 80%) in both CPU and GPU implementations [2]. To effectively handle the huge number of short reads, we modularized the assembly algorithm by focusing on parallelizing the main steps by loading only the necessary data at each stage into the PANDA platform and leaving stage 4 as our future work.

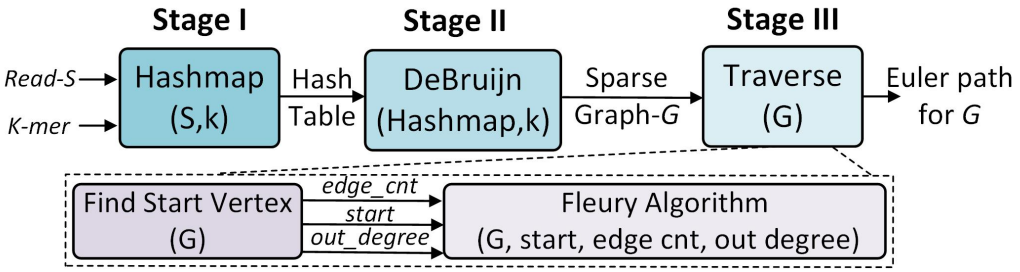


Figure 7. The genome assembly stages.

3.1. Stage One: Hash Table

Algorithm 1 shows the construction of *Hashmap(S,k)* in which the algorithm loops through all the input sequences(S) to generate a mapping of k-mers and its occurances in the genome. For each new k-mer, it creates a hash table entry (key) in the *Hashmap* with frequency = 1 as the initial value. This step is visualized in Figure 8. If the k-mer has been reported previously and is already in the hash table, the frequency is then increased by 1 (New_frq). As indicated, Hashmap procedure can be implemented through *PANDA_Cmp* (comparison), *PANDA_Add* (addition), and *PANDA_Mem_insert* (memory W/R) in-memory operations. Such functions are iteratively used in every step of "for" loop and PANDA is specially designed to handle such computation-intensive load through performing the comparison, summing, and copying operations.

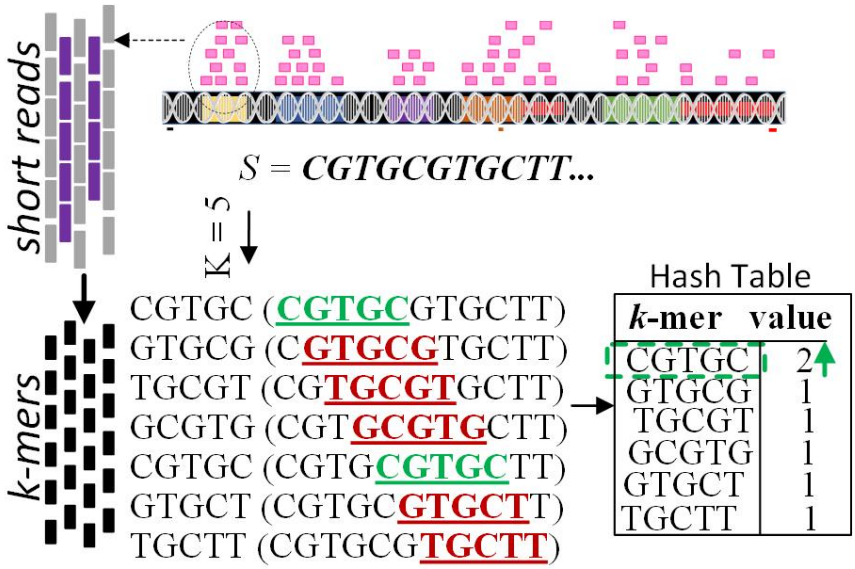


Figure 8. The hash table generation out of k-mers.

Considering the fact that the number of different keys in the hash table is almost comparable to the genome size G, the memory space requirement to save the hash is given by $\sim 2 \times G \times (k+1)$ bits (The factor of 2 is given to represent 2 bits per nucleotide). For instance, storing the hash table for the human genome with $G \sim 3 \times 10^9$ and $k = 32$ requires ~23 GB mostly associated with storing the key. Due to the very large memory space requirement of the hash table for assembly-in-memory algorithm [2], we partition these tables into multiple sub-arrays to fully leverage PANDA's parallelism and to maximize computation throughput. Obviously, larger memory units [38] and distributed memory schemes [2,39] are preferable.

Algorithm 1 Procedure Hashmap (S, k)

Step 1. Initialization:
Hashtable named *Hashmap* = {}
Step 2. Fill out the table:
for i := 0 to length(S)-k+1 **do**
  $k\_mer \leftarrow S[i : i+k]$ ▷ copy values of $S[i$ to $i+k]$ into variable *k_mer*
  if ***PANDA_Cmp***$(k\_mer, Hashmap) == 0$ **then**
    PANDA_Mem_insert$(k\_mer, 1)$
  else
    $New\_frq \leftarrow$ ***PANDA_Add***$(k\_mer, 1)$ ▷ increment frq by 1
    PANDA_Mem_insert(k_mer, New_frq) ▷ insert into Hashmap again
  end if
end for
return Hashmap

The proposed correlated partitioning and mapping methodology, as shown in Figure 9a, locally stores correlated regions of *k*-mer (980 rows) vectors, where each row stores up to 128 bps (*A*,*C*,*G*,*T* encoded by 2 bits) and value (32 rows) vectors in the same sub-array. To count the frequencies of each distinct *k*-mer, the ctrl first reads and parses the short reads from the original sequence bank to the specific sub-array. As depicted in Figure 9a, assuming $S = CGTGTGCA$ as the short read, the *k*-mers- k_i-k_{i+n} are extracted and written into the consecutive memory rows of *k*-mer region. However, when a new query such as k_{i+3} arrives (while k_i-k_{i+2} are already in the memory), it will be first written to the temp region. A parallel in-memory comparison operation (*PANDA_Cmp*) will be performed between temp data and already-stored *k*-mers. Figure 9b intuitively shows *PANDA_Cmp* procedure, where the entire temp row can be compared with a previous *k*-mer row in a single cycle. Then, a built-in ctrl's AND unit in DPU readily takes all the results to determine the next memory operation according to the algorithm. To increase the frequency of a specific *k*-mer, *PANDA_Add* is leveraged to perform in-memory addition without sending data to the off-chip processor.

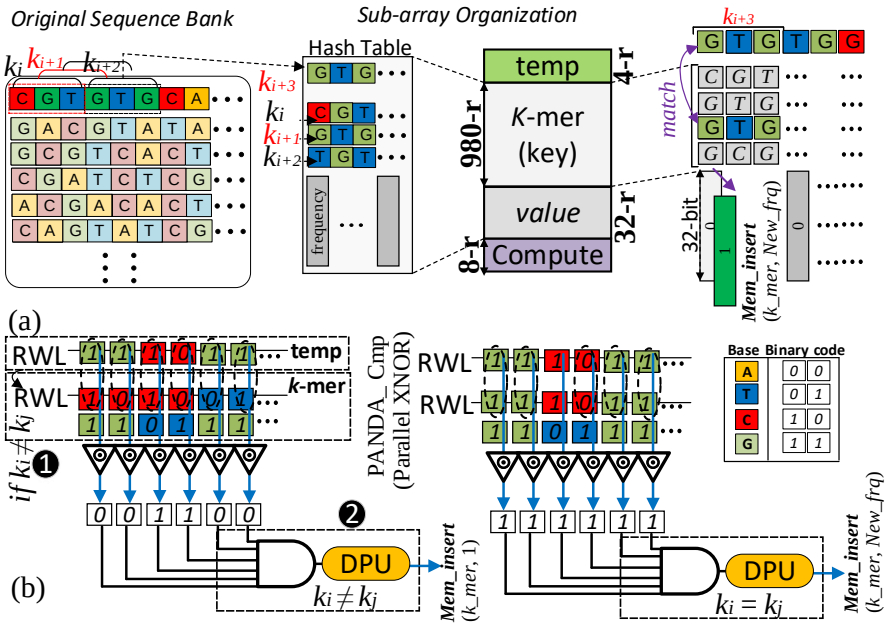


Figure 9. (**a**) The proposed correlated data partitioning and mapping methodology for creating a hash table, (**b**) realization of the parallel in-memory comparator (*PANDA_Cmp*) between *k*-mers in a computational sub-array.

3.2. Stage Two: Graph Construction

The next step is to construct and access the de Bruijn graph based on the pre-generated Hash structure. Each "key" in the Hashmap is a *k*-mer, where the "value" associated with

it is the total occurrence of the k-mer in the genome sequence. To represent a k-mer in the de Bruijn graph, each k-length string will be divided into two nodes: one with the prefix of length $k-1$ and the other with the suffix of length $k-1$ (e.g., CGTGC $\rightarrow$ CGTG and GTGC), and therefore, a directed edge will be connected between the two nodes from left to right. Likewise, k-mers with frequency n will be connected with n edges. The de Bruijn graph G for the sample hash table in Figure 8 is constructed and illustrated in Figure 10 (step 1). Algorithm 2 shows the construction of the de Bruijn graph for PANDA, taking the Hashmap and k as inputs and returning the output graph G. For each key in the hash table, the *PANDA_Mem_insert* instruction creates an entry in G for node1 and node2s. Leveraging adjacency matrix representation for the direct mapping of such a humongous sparse graph into memory comes at a cost of significantly increased memory requirement and run time. The size of the adjacency matrix will be V $\times$ V for any graph with V nodes, where a sparse matrix could be represented by a 3 $\times$ E matrix, where E is the total number of edges in the graph. PANDA utilizes sparse matrix representation shown in Figure 10 (step 2) for mapping purpose. Each entry in the third row of the sparse matrix represents the number of connections between two nodes in the first and second rows.

Algorithm 2 Procedure de Bruijn (Hashmap, k)

Step 1. Initialization:
G = [], $Nodes\_List$ = [], $i = 1$
Step 2. Sparse graph construction:
for $\forall k\_mer \in Hashmap.keys(), i++$ **do**
  $node\_1 \leftarrow k\_mer[0 : k-2]$
  $node\_2 \leftarrow k\_mer[1 : k-1]$
  PANDA_Mem_insert$(G[1][i], node\_1)$
  PANDA_Mem_insert$(G[2][i], node\_2)$
  PANDA_Mem_insert$(G[3][i], Hashmap[k\_mer])$
end for
return G

To balance the workloads of each PANDA's chip and maximize parallelism, we leverage the interval-block partitioning method. We use a hash-based approach [40] by splitting the vertices into M intervals and then dividing edges into M^2 blocks, as shown in Figure 10 (step 3: mapping). Then, each block is allocated to a chip (step 4: allocation) and mapped to its sub-arrays. Having an m-vertex sub-graph with N_s activated sub-arrays (size = $x \times y$), each sub-array can process n vertices ($n \leq f | n \in N, f = min(x, y)$) (step 5: parallel computation). In this way, the number of processing sub-arrays for an N-vertex sub-graph can be formulated as $N_s = \left\lceil \frac{N}{f} \right\rceil$.

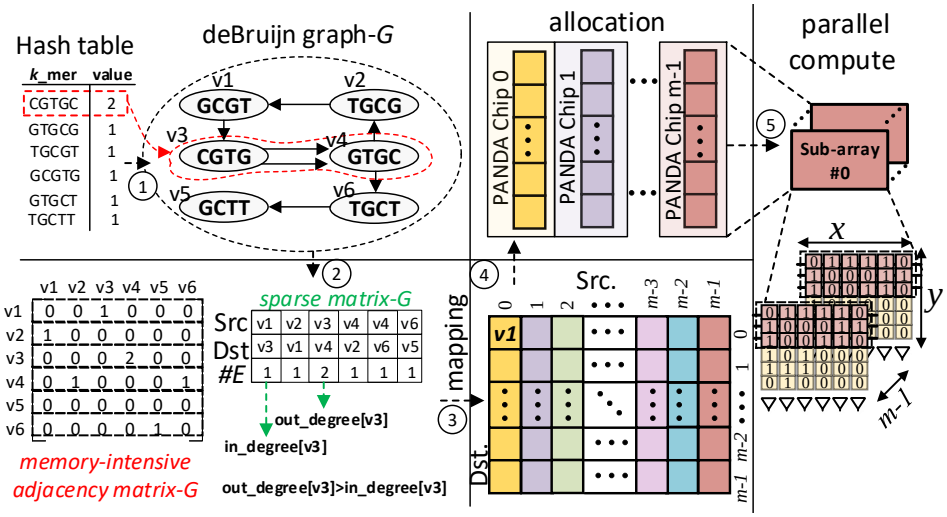


Figure 10. Graph construction with a sparse matrix with partitioning, allocation, and parallel computation.

After graph construction, it is possible to perform a round of simplification on the sparse graph stored in PANDA without the loss of information to avoid fragmentation of the graph. As a matter of fact, the blocks are broken up each time a short read starts or ends leading to linearly connected subgraphs [25]. This fragmentation imposes longer execution times and larger memory spaces. The simplification process easily merges two nodes within memory if node A has only one outgoing edge directed to node B with only one incoming edge.

3.3. Stage Three: Traversal for Euler Path

The input of this stage will be the sparse representation of graph G. In the ideal case, connecting all the edges of the graph G in a continuous manner will reconstruct the entire genome sequence. For traversing all the edges of graph G, we use Fleury's algorithm to find the Euler path of that graph (a path that traverses all the edges of a graph). Basically, a directed graph has an Euler path if the in_degree and out_degree (the in_degree [i] shows how many edges are coming into a vertex-i and out_degree [i] means how many out-going edges vertex-i has) of every vertex is same or, there are exactly two vertices which have $|\text{in\_degree} - \text{out\_degree}| = 1$. Finding the starting vertex is very important to generate the Eulerian path, and we cannot consider any random vertex as the starting. The reconstructed PIM-friendly algorithm for finding the start vertex in graph G is shown in Algorithm 3. For each node, this stage deals with a massive number of iteratively used *PANDA_Add* to calculate the number of in_degree, out_degree, and edge_cnt (total number of edges). Moreover, in order to check the condition ($|\text{out\_degree} = \text{in\_degree}| + 1$), a parallel *PANDA_Cmp* operation is required.

Algorithm 3 Procedure find start vertex (G)

Step 1. Initialization:
$start \leftarrow 0, end \leftarrow 0$
$edge\_cnt \leftarrow 0$ ▷ For counting number of edges in G
$Len \leftarrow size(G)$
Step 2. Find the start vertex:
for n in Nodes **do**
  $in\_degree[i] \leftarrow 0$
  $out\_degree[i] \leftarrow 0$
end for
for n in Nodes **do**
  for k :=1 to Len **do**
    if ***PANDA_Cmp***$(G[1][k], n)$ **then** ▷ node n has an out-going edge
      $out\_degree[n] \leftarrow$ ***PANDA_Add***$(out\_degree[n], int(G[3][k]))$
      $in\_degree[int(G[2][k])] \leftarrow$ ***PANDA_Add***$(in\_degree[int(G[2][k])], int(G[3][k]))$
      $edge\_cnt \leftarrow$ ***PANDA_Add***$(edge\_cnt, int(G[3][k]))$
    end if
  end for
  if ***PANDA_Cmp***$(out\_degree[n], in\_degree[n] + 1)$ **then**
    $start \leftarrow n$
  else
    $start \leftarrow first\_node$
  end if
end for
return start and edge_cnt and out_degree

After finding the start node, PANDA has to traverse through the length of the sparse matrix G from the starting vertex, checking the two aforementioned conditions for each edge and accordingly adding qualified edges to the Eulerian path. Algorithm 4 shows the reconstructed Fleury algorithm. If an edge *is not a bridge* (removing the edge will disconnect the graph into two parts) and *is not the last edge of the graph*, edge $(start, v)$ is added in the Eulerian path, and the edge will be removed from the graph. $isValidNextEdge()$ function will check if the edge (u, v) is valid to be included into the Eulerian path. If v is the only adjacent vertex remaining for u, it means all other adjacent vertices have been traversed already, so this edge could be counted now. The second condition counts the number

of reachable nodes from u before and after removing the edge. If the number changes or decreases, it means the edge was a bridge previously, so it cannot be removed from the graph.

Algorithm 4 Procedure Fleury (G, node, edge_count, out_degree)

```
for v := 0 to N do
    if G[1][k] == start then
        v ← G[2][k]
        if isValidNextEdge(v) then
            PANDA_Mem_insert(v)                    ▷ add (start, v) in the Eulerian path
            PANDA_Add(out_degree[start], −1)
            PANDA_Add(G[3][k], −1)                 ▷ remove one edge from the graph
            PANDA_Add(edge_cnt, −1)
        end if
    end if
    Fleury(G, v, edge_count, out_degree[])         ▷ run Fleury again for the next node v
end for
```

In the interest of space, we show out_/in_degree and edge_cnt mapping and computation in the PANDA platform in Figure 11, which basically sums up all the entries of a particular node i of valid links connected to a vertex to find the start vertex. As can be seen, we use the sparse matrix representation to store the matrix-G. In our mapping technique, each column is assigned to a distinct source vertex in the graph and then filled out with the number of edges (#E) only linked to existing destination vertices in a vertical fashion. Therefore, we do not assign destination vertices to the memory rows as in direct adjacency matrix mapping. Here, we consider a 4-bit representation for the simplicity. For example, v4 has outgoing edges to v2 and v6 that are stored vertically in a sub-array. PANDA could perform parallel in-memory addition to calculate the total number of out_ degrees for all nodes in parallel. For this task, two rows in the sub-array are initialized to zero as Carry reserved rows such that they can be selected along with two operands (here v4 → v2 data (0001) and v4 → v6 data (0001)) to perform parallel in-memory addition. To perform parallel addition operation and generate initial Carry and Sum bits, PANDA takes every three rows to perform a parallel in-memory addition. The results are written back to the memory reserved space (Resv.). Then, the next step only deals with the multi-bit addition of resultant data starting bit-by-bit from the LSBs of the two words and continuing towards MSBs. Then PANDA is able to perform a comparison between a number of out_degree and in_degree for each node in parallel to determine the start node. After finding the start node as shown in Figure 11, contig. generation can be readily accomplished by finding the Eulerian path and putting together each vertex data from different sub-arrays.

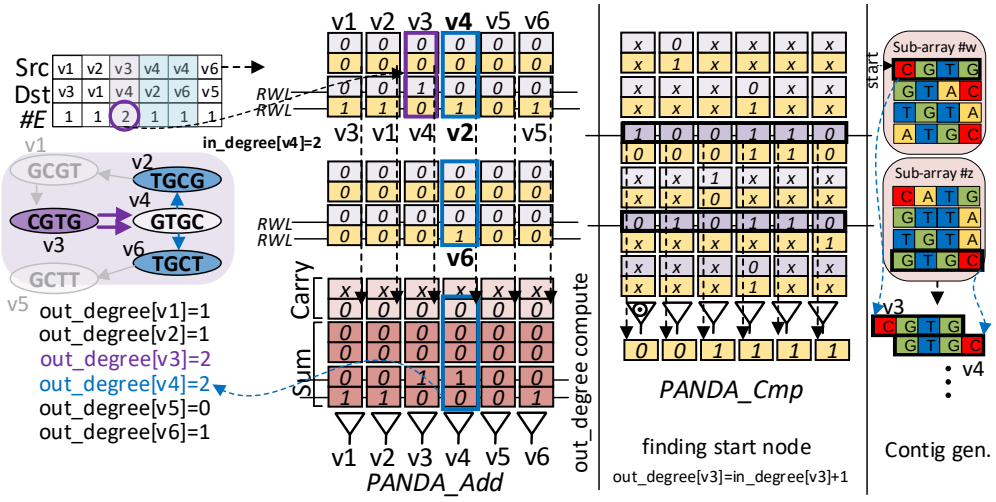


Figure 11. PANDA in-memory addition and comparison scheme for finding the start vertex.

4. Performance Estimation

4.1. Setup

Accelerator: To the best of our knowledge, this work is the first to explore the performance of a PIM platform for genome assembly problems, therefore, we have to create the evaluation test bed from scratch to have an impartial comparison with both von Neumann and non-von Neumann architectures. We configure the PANDA's computational memory sub-array with 1024 rows and 256 columns, 4 × 4 memory matrix (with 1/1 as row/column activation) per bank organized in H-tree routing manner, 16 × 16 banks (with 1/1 as row/column activation) in each memory chip. For comparison, we consider five computing platforms: (1) A general purpose processor (GPP): a Quad-Core Intel Core i7-7700 CPU @ 3.60 GHz processor with 8192MB DIMM DDR4 1600MHz RAM and 8192KB Cache; (2) A processing in STT-MRAM platform capable of performing bulk bit-wise operations [41]; (3) A recently developed processing in SOT-MRAM platform for DNA sequence alignment optimized to perform comparison-intensive operations [6]; (4) A processing in ReRAM accelerator designed for accelerating bulk bit-wise operations [42]; (5) A processing in DRAM accelerator based on Ambit [12] working with triple row activation mechanism to implement various functions. The detailed evaluation framework developed for PIM platforms is shown in Figure 12. All PIM platforms have an identical physical memory configuration as PANDA. Additionally, we developed a similar cross-layer simulation framework starting from device-level simulation all the way to circuit- and architectural level as explained for PANDA in Section 2.4. The results of the architecture evaluation of all PIM platforms were then fed to a high-level in-house simulator developed in Matlab to perform each genome assembly stage based on our customized and PIM-friendly algorithm and estimate the overall performance. It is noteworthy that DPU was developed in HDL and the performance results were extracted with Synopsys design compiler [43] and fed to the developed NVSim library for each PIM platform.

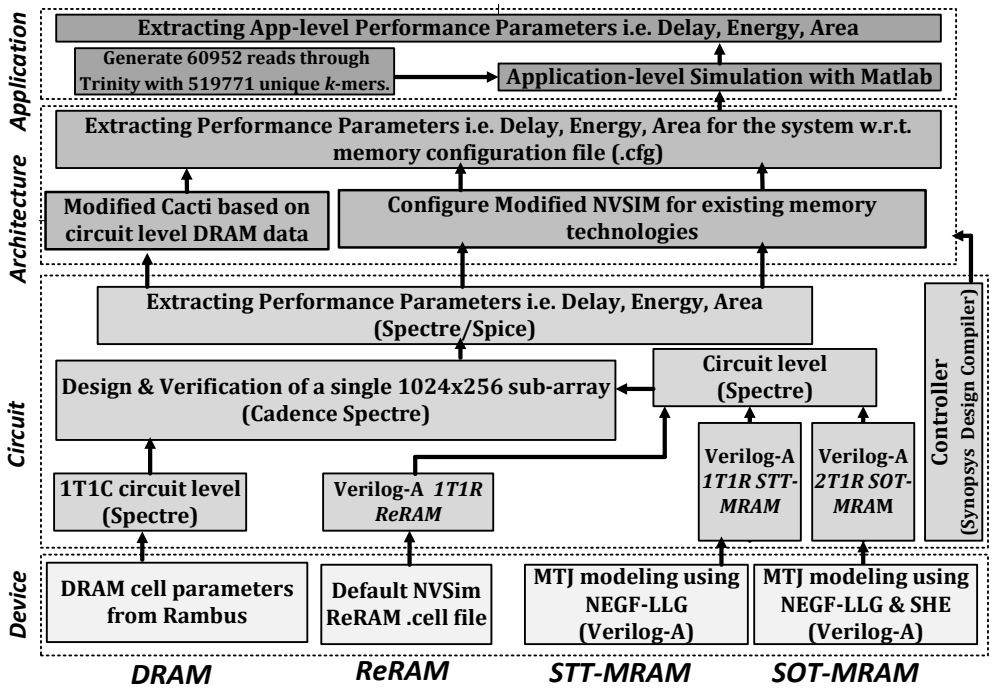


Figure 12. Evaluation framework developed for processing in-memory platforms.

To evaluate the CPU performance, we use Trinity-v2.8.5 [26] which was shown to be sensitive and efficient in recovering full-length transcripts. Trinity constructs the de Bruijn graph from short-read sequences employs an enumeration algorithm to score all branches, and keeps possible ones as isoforms/transcripts.

Experiment: In our experiment, we create 60,952 short reads through the Trinity sample genome bank with 519,771 unique *k*-mers. We initially set the *k*-mer length, *k*, to default 25, and then change it to 22, 27, and 32 as typical values for most genome assemblers. To clarify, the CPU executes the *Inchworm*, *Chrysalis*, and *Butterfly* steps in Trinity, while PIM platforms run three main procedures in genome assembly shown in Figure 7, i.e., Hashmap, DeBruijn, and Traverse for under-test PIM platforms. We compare Trinity's power consumption and execution time to that of other PIM assemblers by several measures. To have a fair comparison with such a comprehensive assembler (that performs full genome assembly task with scaffolding step), we penalized the PIM platforms with ∼25% excessive time and power. We believe this could provide a more realistic comparison with a von Neumann architecture-based assembler. We developed a high-level compiler program connected to our architectural simulator to perform a preprocessing step and essentially convert each step of the customized algorithms (Algorithms 1–4) to the corresponding predefined ISA shown in Section 2. As shown in previous works, such designs could efficiently solve the operand locality issue in PIM platforms. The developed framework initially traverses through the original sequence bank and organizes the k-mers before mapping into the sub-array. Accordingly, the k-mers are mapped and aligned in the computational sub-array with respect to the user's provided configuration file. In this way, we ensure that k-mers and values are correctly aligned in the memory before the computation step.

4.2. Run Time

The execution time of genome assembly tasks for different platforms is reported in Figure 13. For k = 25, the CPU platform executes the *Inchworm*, *Chrysalis*, and *Butterfly* steps [26] of Trinity in ∼32 s, where Chrysalis for clustering the contigs and constructing complete de Bruijn graph takes the largest fraction of run time (28 s) as expected. However, the comparison operation-intensive Hashmap procedure for *k*-mer analysis takes the largest fraction of execution time across all PIM platforms (over 40% of total run time). A larger *k*-mer length typically diminishes the de Bruijn graph connectivity by simultaneously reducing the number of ambiguous repeats in the graph and the chance of overlap between two reads. This is why the run time for all platforms reduces with an increase in *k*-mer length.

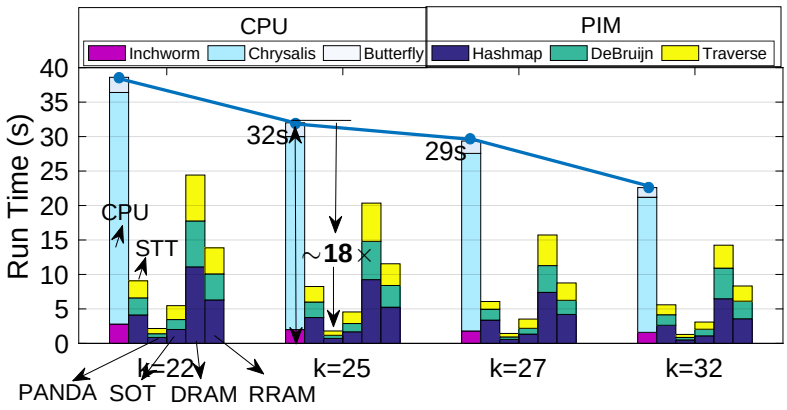


Figure 13. The breakdown of run time for under-test platforms running different *k*-mer length genome assembly task. In each bar group from left to right: CPU, processing in STT-MRAM [41], PANDA, processing in SOT-MRAM [6], processing in DRAM [12], and processing in RRAM [42].

We can observe that PIM platforms reduce the run time remarkably compared to the CPU. As shown, PANDA reduces the run time by ∼18× compared to the CPU platform for k = 25 (18.8× on average over four different *k*-mer lengths). The PANDA platform essentially accelerates the graph construction and traversal stages by ∼21.5× compared with the CPU platform. Here, by increasing the *k*-length to 32, a higher speed-up is even achievable. Compared with counterpart PIM platforms, our X(N)OR-friendly de-

sign reduces the run time on average by 4.2× and 2.5× compared to the STT-PIM [41] and SOT-PIM [6] platforms, respectively, as the fastest counterparts. This comes from the fact that under-test PIM platforms require multi-cycle operations to implement additional operations. Moreover, the SOT-based device intrinsically shows higher write speed compared to STT devices. Compared to DRAM and RRAM platforms, PANDA achieves on average 10.9 × and 6 × speed-up for various length *k*-mer processing. It is worth pointing out that the processing in DRAM platforms possesses a destructive computing operation and requires multiple memory cycles to copy the operands to particular rows before computation. As for Ambit [12], seven memory cycles are needed to implement the in-memory-X(N)OR function.

4.3. Power Consumption

We estimated the power consumption of different PIM platforms for running different length *k*-mers compared to the CPU platform as shown in Figure 14. Based on our results, a significant reduction in power consumption can be reported for all under-test PIM platforms compared with the CPU. The breakdown of energy consumption is also shown for the PIM platforms; however, this could not be accurately achieved for the CPU and overall power consumption is reported. In our experiment, processing in SOT-MRAM design [6] achieves the smallest power consumption (on average) to run the three main procedures, as compared with the CPU and other PIM platforms. The PANDA platform stands as the second most power-efficient design. This is mainly due to the three-SA-based bit-line computing scheme in PANDA compared with the two-SA per bit-line technique in the counterpart design; while the proposed scheme brings more speed-up compared with the design in [6], it requires relatively more power. PANDA reduces power consumption by ∼9.2× on average compared with the CPU platform over different length *k*-mers. Moreover, it reduces the power consumption by ∼18% compared with the STT-MRAM [41] platform. The main reason behind this improvement is more efficient addition operation in PANDA. The addition operation requires additional memory cycles in the STT-MRAM [41] platform to save and carry the bit back to the memory and use it again for the computation of the next bits. Compared to DRAM and RRAM platforms, PANDA obtains on average 2.11× and 55% power reduction for various length *k*-mer processing.

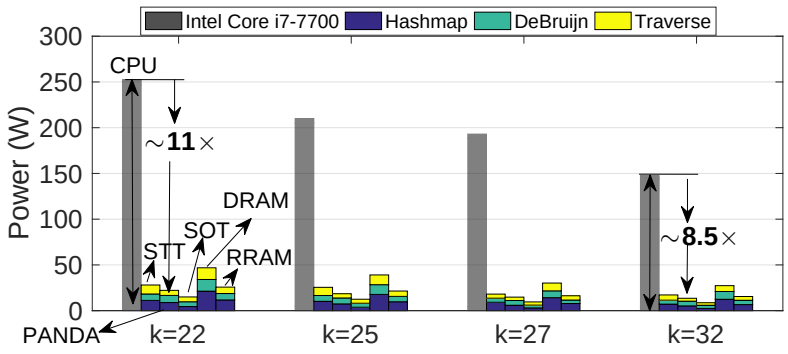


Figure 14. The breakdown of power consumption for PIM platforms running different *k*-mer length genome assembly task compared to CPU. In each bar group from left to right: CPU, processing in STT-MRAM [41], PANDA, processing in SOT-MRAM [6], processing in DRAM [12], and processing in RRAM [42].

We estimated the energy consumption imposed by the sensing circuit at the sub-array level in order to analyze the adverse impacts of boosting the sense current for read, 2-input, and 3-input PIM operation as a follow-up discussion to the reliability part in Section 2.4. Figure 15 shows that by increasing the sense current, the sub-array's dynamic energy will increase correspondingly. Our experiment shows that to increase the average sense margin by ∼25–38 mv, the PANDA's computational sub-array consumes ∼1.5× more energy. This could be considered a significant factor in designing sensing circuitry.

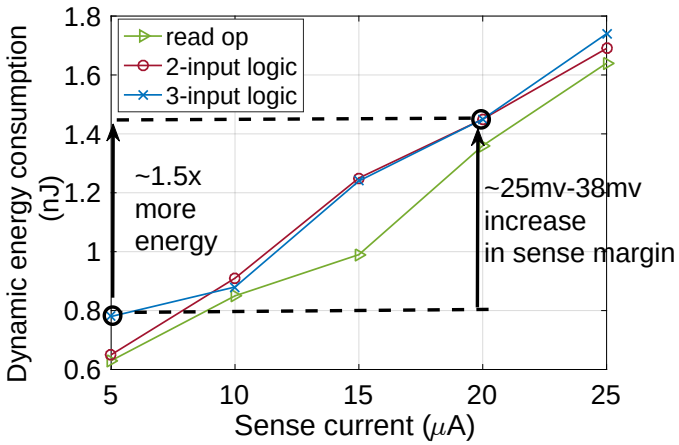


Figure 15. The dynamic energy consumption trade-off with the sense current as a countermeasure to improve the sense margin in PANDA.

4.4. Speed-Up/Power Efficiency Trade-Off

We investigate the power efficiency and speed-up of the three best under-test PIM platforms, based on the run time and power consumption results in the previous sub-sections, by tuning the number of active sub-arrays (N_s) associated with the comparison and addition operations. A parallelism degree (P_d) can then be defined as the number of replicated sub-arrays to boost the performance of the PIM platforms through parallel processing, as shown in prior works [6,15]. For example, when P_d is set to 2, two parallel sub-arrays are undertaken to process the in-memory operations, simultaneously. We expect such parallelism to improve the performance of genome assembly at the cost of sacrificing the power consumption and area. Figure 16 plots the existing trade-off between run time and power consumption vs. P_d for $k = 25$. The estimated CPU power budget required to execute Trinity is also shown. It can be seen that for all platforms the run time reduces by increasing the parallelism. For example, for the PANDA platform in an extreme case, increasing P_d from 1 to 8 increases the power consumption from ∼19 W to 128 W (∼7×) and reduces the execution time by a factor of 3, which might not be a favorable case. Therefore, a user can meticulously tailor the PANDA performance to meet the system/application constraints. Here, we show the optimum theoretical performance of PANDA and other PIM platforms by pinpointing the intersection between power and run time curves in Figure 16. We observe that PANDA achieves the smallest run time and power consumption task with a $P_d \sim 2$ compared with the others.

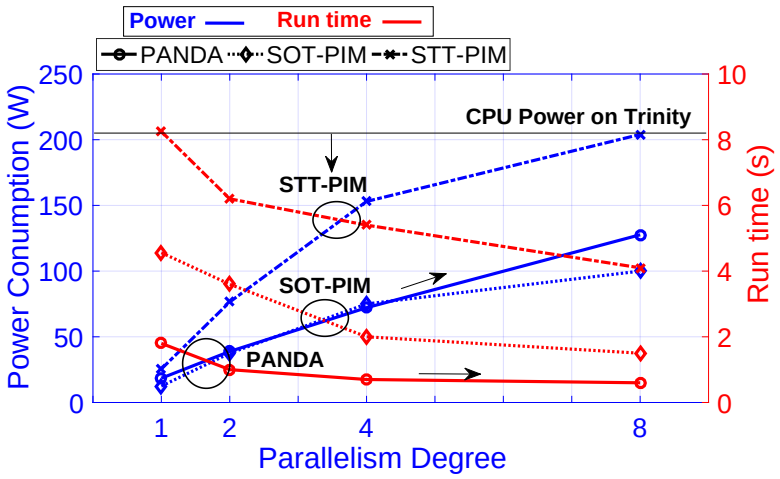


Figure 16. Trade-off between power consumption and run-time with respect to parallelism degree in $k = 25$.

4.5. Memory Wall Challenge

The power efficiency and speed-up of PIM platforms against the von Neumann architecture-based CPU were discussed in prior subsections. Here, we further explore the reasons behind the numbers reported by considering two new measures, i.e., Memory Bottleneck Ratio (MBR) and Resource Utilization Ratio (RUR). We define MBR as the time fraction needed for data transfer from/to on-chip or off-chip when the computation has to wait for data, i.e., memory wall happens. We also define RUR as the time fraction in which the computation resources are loaded with data. The memory wall is considered as the main bottleneck that brings large power consumption and lengthens execution time in the CPU. The MBR is reported in Figure 17a. The peak throughput for each design in four distinct *k*-mer lengths is taken into account for performing the evaluation. This evaluation mainly considers the number of memory accesses. As shown, the PANDA uses less than ∼17% time for data transfer due to the PIM acceleration schemes, while CPU's MBR increases to 65% when $k = 25$. Moreover, we observe that all the other PIM platforms except DRAM also spend less than ∼17% time for data communication. The smaller MBR can be translated as the higher RUR for the accelerators plotted in Figure 17b. The lower MBR can be understood as a higher RUR. We can see that with up to ∼82%, PANDA achieves the highest RUR. Taking everything into account, PIM acceleration schemes offer a high utilization ratio (>60% excluding DRAM) confirming the conclusion drawn in Figure 17a. The memory wall evaluation shows the efficiency of the PANDA platform for solving the memory wall challenge.

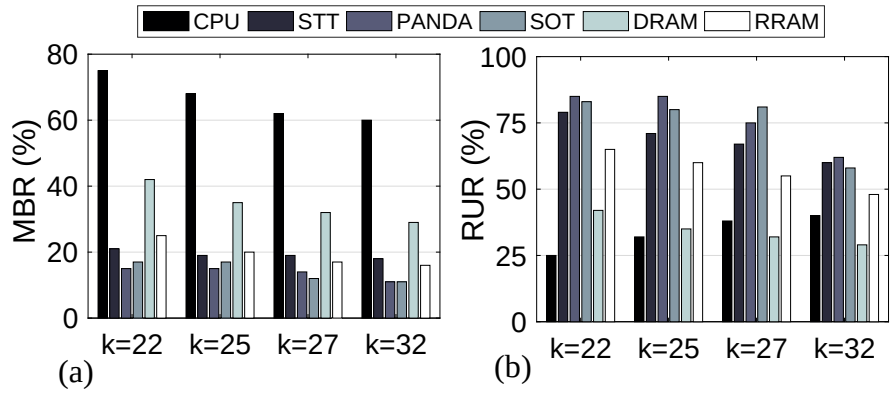


Figure 17. (**a**) The memory bottleneck ratio and (**b**) resource utilization ratio for CPU and three under-test PIM platforms for running genome assembly tasks.

4.6. Area Overhead

To estimate the area overhead of PANDA on top of the original MRAM die, three crucial hardware cost sources must be taken into consideration. First, the controller unit is located in the sub-array and MAT level; second, add-on WBL and RBL voltage drivers and peripherals; and third, add-on transistors to SAs to enable in-memory computing. Figure 18a depicts the breakdown of the area overhead resulting from add-on hardware to original PANDA memory at MAT level. Our experiments show that, in total, PANDA imposes ∼7.9% area overhead to the memory die, where the modified controller, drivers, and then SA contribute more than 70% of this area overhead. We also calculated the area overhead of the processing-SOT-MRAM platform in [6] as the most similar counterpart implemented with the same technology, as shown in Figure 18b. Please note that we considered the area overhead of DPU for both designs as well. This design incurs a smaller overhead (5.9%) to the memory die.

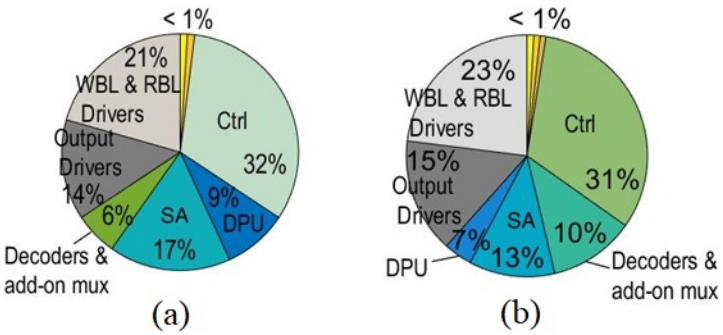


Figure 18. The breakdown of area-overhead in MAT level for (**a**) PANDA and (**b**) processing in SOT-MRAM platform [6].

5. Conclusions

The processing-in-memory paradigm has emerged as an efficient computing approach for analyzing large-scale data, including DNA sequences. In this paper, we presented PANDA as a new processing-in-SOT-MRAM platform to accelerate the comparison/addition-extensive genome assembly application using PIM-friendly operations. We developed PANDA based on a set of new circuit-level schemes to realize a data-parallel computational core for genome assembly. The platform is configured with a novel data partitioning and mapping technique that provides local storage and processing to fully utilize our customized algorithm-level parallelism. The cross-layer simulation results demonstrate that PANDA reduces the execution time and power by ∼18× and ∼11×, respectively, compared with the CPU. Moreover, speed-ups of up to 2–4× can be obtained over recent processing in MRAM platforms to perform a similar task. Future endeavors can take diverse paths. Firstly, delving into different innovative PIM platforms to enhance the speed of each phase of DNA assembly. Secondly, developing new hardware mapping and processing techniques that are correlated with minimizing the overhead associated with intermediate results write-backs. Thirdly, introducing algorithmic innovations in graph construction to further enhance the performance of the accelerator.

Author Contributions: Investigation, S.A., N.A.F., W.Z. and D.F.; Methodology, S.A., N.A.F. and D.N.; Supervision, W.Z. and D.F.; Visualization, S.A. and D.N.; Writing—original draft, S.A. and N.A.F.; Writing—review & editing, S.A. and D.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported in part by the National Science Foundation under Grant No. 2349802 and No. 2314591.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Li, H.; Homer, N. A survey of sequence alignment algorithms for next-generation sequencing. *Brief. Bioinform.* **2010**, *11*, 473–483. [CrossRef] [PubMed]
2. Georganas, E.; Buluç, A.; Chapman, J.; Oliker, L.; Rokhsar, D.; Yelick, K. Parallel de bruijn graph construction and traversal for de novo genome assembly. In Proceedings of the SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; IEEE: New York, NY, USA, 2014; pp. 437–448.
3. Sinha, A.; Yang, H.-C.; Liu, P.-Y.; Kuo, Y.-S.; Fang, Y.; Chang, T.-S.; Li, K.-H.; Lai, B.-C. DSIM: Distributed Sequence Matching on Near-DRAM Accelerator for Genome Assembly. *J. Emerg. Sel. Top. Circuits Syst.* **2022**, *12*, 486–499. [CrossRef]
4. Chapman, J.A.; Ho, I.; Sunkara, S.; Luo, S.; Schroth, G.P.; Rokhsar, D.S. Meraculous: De novo genome assembly with short paired-end reads. *PLoS ONE* **2011**, *6*, e23501. [CrossRef] [PubMed]
5. Zokaee, F.; Zarandi, H.R.; Jiang, L. Aligner: A process-in-memory architecture for short read alignment in rerams. *IEEE Comput. Archit. Lett.* **2018**, *17*, 237–240. [CrossRef]

6. Angizi, S.; Sun, J.; Zhang, W.; Fan, D. AlignS: A processing-in-memory accelerator for DNA short read alignment leveraging SOT-MRAM. In Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2–6 June 2019; pp. 1–6.
7. Shahroodi, T.; Miao, M.; Lindegger, J.; Wong, S.; Mutlu, O.; Hamdioui, S. An In-Memory Architecture for High-Performance Long-Read Pre-Alignment Filtering. *arXiv* **2023**, arXiv:2310.15634.
8. Rumpf, M.D.; Alser, M.; Gollwitzer, A.E.; Lindegger, J.; Almadhoun, N.; Firtina, C.; Mangul, S.; Mutlu, O. SequenceLab: A Comprehensive Benchmark of Computational Methods for Comparing Genomic Sequences. *arXiv* **2023**, arXiv:2310.16908.
9. De Sandre, G.; Bettini, L.; Pirola, A.; Marmonier, L.; Pasotti, M.; Borghi, M.; Mattavelli, P.; Zuliani, P.; Scotti, L.; Mastracchio, G.; et al. A 90 nm 4 Mb embedded phase-change memory with 1.2 V 12 ns read access time and 1MB/s write throughput. In Proceedings of the 2010 IEEE International Solid-State Circuits Conference-(ISSCC), San Francisco, CA, USA, 7–11 February 2010.
10. Tsuchida, K.; Inaba, T.; Fujita, K.; Ueda, Y.; Shimizu, T.; Asao, Y.; Kajiyama, T.; Iwayama, M.; Sugiura, K.; Ikegawa, S.; et al. A 64 Mb MRAM with clamped-reference and adequate-reference schemes. In Proceedings of the 2010 IEEE International Solid-State Circuits Conference-(ISSCC), San Francisco, CA, USA, 7–11 February 2010.
11. Chang, M.F.; Shen, S.J.; Liu, C.C.; Wu, C.W.; Lin, Y.F.; King, Y.C.; Yamauchi, H. An Offset-Tolerant Fast-Random-Read Current-Sampling-Based Sense Amplifier for Small-Cell-Current Nonvolatile Memory. *J. Solid-State Circuits* **2013**, *48*, 864–877. [CrossRef]
12. Seshadri, V.; Lee, D.; Mullins, T.; Hassan, H.; Boroumand, A.; Kim, J.; Kozuch, M.A.; Mutlu, O.; Gibbons, P.B.; Mowry, T.C. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, USA, 14–18 October 2017; pp. 273–287.
13. Yu, J.; Nane, R.; Ashraf, I.; Taouil, M.; Hamdioui, S.; Corporaal, H.; Bertels, K. Skeleton-based Synthesis Flow for Computation-In-Memory Architectures. *IEEE Trans. Emerg. Top. Comput.* **2017**, *2*, 545–558. [CrossRef]
14. Zhang, F.; Angizi, S.; Fahmi, N.A.; Zhang, W.; Fan, D. PIM-Quantifier: A Processing-in-Memory Platform for mRNA Quantification. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 43–48.
15. Li, S.; Xu, C.; Zou, Q.; Zhao, J.; Lu, Y.; Xie, Y. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In Proceedings of the 53rd Annual Design Automation Conference, New York, NY, USA, 5–9 June 2016; pp. 1–6.
16. Chowdhury, Z.I.; Zabihi, M.; Khatamifard, S.K.; Zhao, Z.; Resch, S.; Razaviyayn, M.; Wang, J.P.; Sapatnekar, S.S.; Karpuzcu, U.R. A DNA Read Alignment Accelerator Based on Computational RAM. *IEEE J. Explor. Solid-State Comput. Devices Circuits* **2020**, *6*, 80–88. [CrossRef]
17. Kang, W.; Wang, H.; Wang, Z.; Zhang, Y.; Zhao, W. In-memory processing paradigm for bitwise logic operations in STT–MRAM. *IEEE Trans. Magn.* **2017**, *53*, 1–4.
18. Angizi, S.; Sun, J.; Zhang, W.; Fan, D. GraphS: A graph processing accelerator leveraging SOT-MRAM. In Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 378–383.
19. Li, R.; Yu, C.; Li, Y.; Lam, T.W.; Yiu, S.M.; Kristiansen, K.; Wang, J. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics* **2009**, *25*, 1966–1967. [CrossRef] [PubMed]
20. Richard Wilton, A.S.S. Performance optimization in DNA short-read alignment. *Bioinformatics* **2022**, *38*, 2081–2087. [CrossRef] [PubMed]
21. Liu, C.M.; Wong, T.; Wu, E.; Luo, R.; Yiu, S.M.; Li, Y.; Wang, B.; Yu, C.; Chu, X.; Zhao, K.; et al. SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics* **2012**, *28*, 878–879. [CrossRef] [PubMed]
22. Arram, J.; Kaplan, T.; Luk, W.; Jiang, P. Leveraging FPGAs for accelerating short read alignment. *IEEE/ACM Trans. Comput. Biol. Bioinform.* **2016**, *14*, 668–677. [CrossRef] [PubMed]
23. Mahmood, S.F.; Rangwala, H. Gpu-euler: Sequence assembly using gpgpu. In Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications, Banff, AB, Canada, 2–4 September 2011; pp. 153–160.
24. Varma, B.S.C.; Paul, K.; Balakrishnan, M. FPGA-Based Acceleration of De Novo Genome Assembly. In *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 55–79.
25. Zerbino, D.R.; Birney, E. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* **2008**, *18*, 821–829. [CrossRef]
26. Grabherr, M.G.; Haas, B.J.; Yassour, M.; Levin, J.Z.; Thompson, D.A.; Amit, I.; Adiconis, X.; Fan, L.; Raychowdhury, R.; Zeng, Q.; et al. Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nat. Biotechnol.* **2011**, *29*, 644–652. [CrossRef] [PubMed]
27. Goswami, S.; Lee, K.; Shams, S.; Park, S.J. Gpu-accelerated large-scale genome assembly. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 21–25 May 2018; pp. 814–824.
28. Ren, S.; Ahmed, N.; Bertels, K.; Al-Ars, Z. An Efficient GPU-Based de Bruijn Graph Construction Algorithm for Micro-Assembly. In Proceedings of the 2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE), Taichung, Taiwan, 29–31 October 2018; pp. 67–72.
29. Lu, M.; Luo, Q.; Wang, B.; Wu, J.; Zhao, J. GPU-accelerated bidirected De Bruijn graph construction for genome assembly. In Proceedings of the Asia-Pacific Web Conference, Sydney, Australia, 4–6 April 2013; pp. 51–62.

30. Angizi, S.; Fahmi, N.A.; Zhang, W.; Fan, D. PIM-Assembler: A processing-in-memory platform for genome assembly. In Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 20–24 July 2020; pp. 1–6.
31. Angizi, S. Processing-in-Memory for Data-Intensive Applications, from Device to Algorithm. Ph.D. Thesis, Arizona State University, Tempe, AZ, USA, 2021.
32. Fong, X.; Kim, Y.; Yogendra, K.; Fan, D.; Sengupta, A.; Raghunathan, A.; Roy, K. Spin-transfer torque devices for logic and memory: Prospects and perspectives. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2015**, *35*, 1–22. [CrossRef]
33. Pai, C.F.; Liu, L.; Li, Y.; Tseng, H.; Ralph, D.; Buhrman, R. Spin transfer torque devices utilizing the giant spin Hall effect of tungsten. *Appl. Phys. Lett.* **2012**, *101*, 122404. [CrossRef]
34. Razavi, B. The StrongARM latch [a circuit for all seasons]. *IEEE Solid-State Circuits Mag.* **2015**, *7*, 12–17. [CrossRef]
35. Yuasa, S.; Nagahama, T.; Fukushima, A.; Suzuki, Y.; Ando, K. Giant room-temperature magnetoresistance in single-crystal Fe/MgO/Fe magnetic tunnel junctions. *Nat. Mater.* **2004**, *3*, 868. [CrossRef]
36. Mutlu, O.; Ghose, S.; Gómez-Luna, J.; Ausavarungnirun, R. A Modern Primer on Processing in Memory. *arXiv* **2020**, arXiv:2012.03112.
37. Patel, M.; Kim, J.S.; Hassan, H.; Mutlu, O. Understanding and modeling on-die error correction in modern DRAM: An experimental study using real devices. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 24–24 June 2019; pp. 13–25.
38. Li, R.; Zhu, H.; Ruan, J.; Qian, W.; Fang, X.; Shi, Z.; Li, Y.; Li, S.; Shan, G.; Kristiansen, K.; et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.* **2010**, *20*, 265–272. [CrossRef]
39. Simpson, J.T.; Wong, K.; Jackman, S.D.; Schein, J.E.; Jones, S.J.; Birol, I. ABySS: A parallel assembler for short read sequence data. *Genome Res.* **2009**, *19*, 1117–1123. [CrossRef] [PubMed]
40. Dai, G.; Huang, T.; Chi, Y.; Zhao, J.; Sun, G.; Liu, Y.; Wang, Y.; Xie, Y.; Yang, H. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *38*, 640–653. [CrossRef]
41. Jain, S.; Ranjan, A.; Roy, K.; Raghunathan, A. Computing in memory with spin-transfer torque magnetic RAM. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *26*, 470–483. [CrossRef]
42. Imani, M.; Kim, Y.; Rosing, T. Mpim: Multi-purpose in-memory processing using configurable resistive memory. In Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, Japan, 16–19 January 2017; pp. 757–763.
43. Synopsys Inc. *Synopsys Design Compiler, Product Version 14.9.2014*; Synopsys Inc.: Sunnyvale, CA, USA, 2014.