PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly

Shaahin Angizi*, Naima Ahmed Fahmi†, Wei Zhang† and Deliang Fan*
*School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ
†Department of Computer Science, University of Central Florida, Orlando, FL
Email: sangizi@asu.edu, dfan@asu.edu

***Abstract*—In this paper, for the first time, we propose a high-throughput and energy-efficient Processing-in-DRAM-accelerated genome assembler called *PIM-Assembler* based on an optimized and hardware-friendly genome assembly algorithm. PIM-Assembler can assemble large-scale DNA sequence dataset from all-pair overlaps. We first develop *PIM-Assembler* platform that harnesses DRAM as computational memory and transforms it to a fundamental processing unit for genome assembly. *PIM-Assembler* can perform efficient X(N)OR-based operations inside DRAM incurring low cost on top of commodity DRAM designs (∼5% of chip area). *PIM-Assembler* is then optimized through a correlated data partitioning and mapping methodology that allows local storage and processing of DNA short reads to fully exploit the genome assembly algorithm-level's parallelism. The simulation results show that *PIM-Assembler* achieves on average 8.4× and 2.3× higher throughput for performing bulk bit-wise XNOR-based comparison operations compared with CPU and recent processing-in-DRAM platforms, respectively. As for comparison/addition-extensive genome assembly application, it reduces the execution time and power by ∼5× and ∼7.5× compared to GPU.**

I. Introduction

Advances in high-throughput sequencing technologies have enabled accurate and fast generation of large-scale genomic data for each individual, and is capable of measuring molecular activities in cells. Genomic analyses, including mRNA quantification, genetic variants detection, and differential gene expression analysis, promise to help improve phenotype predictions and provide more accurate disease diagnostics [1]. Genome assembly refers to the process of taking a large number of short DNA reads and putting them back together to realize the original chromosomes from which the DNA originated. The ultimate goal of a sequence assembler is to produce long contiguous pieces of sequence (contigs) from these short reads since the current DNA sequencing technology cannot read whole genomes in one step [2].

Today's bioinformatics application acceleration solutions are mostly based on the Von-Neumann architecture with separate computing and memory components connecting via buses and inevitably consumes a large amount of energy in data movement between them [3]. 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 [4], [5] and especially processing-in-DRAM architecture has achieved remarkable success by dramatically reducing data transfer energy and latency [3], [5], [6]. 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. Besides, most of CPU [7]-/ GPU [8]-/ FPGA [9]- and even PIM [4]-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 [10]. There are multiple CPU-based genome assembly tools implementing the bidirected deBruijn graph model for genome assembly such as Velvet [11], etc. However, only a few GPU-accelerated genome assemblers have been presented such as GPU-Euler [10]. This mainly comes from the nature of the assembly workload that is not only compute-intensive but also extremely data-intensive requiring very large memories. Therefore adapting such problem to use GPUs with their limited memory capacities has brought many challenges [12]. These bottlenecks motivate us to show that the genome assembly problem can exploit the large internal bandwidth of DRAM chip for PIM acceleration. Moreover, with a careful observation of genome assembly workload, it turns out this task heavily relies on comparison and addition operations. However, due to the intrinsic complexity of X(N)OR logic, the throughput of processing-in-DRAM platforms [3], [5], [6], [13] unavoidably diminishes when dealing with such bulk bit-wise operations. This is because majority/AND/OR-based multi-cycle operations and required row initialization in the previous designs [3], e.g., Ambit [5] imposes 7 memory cycles to implement X(N)OR logic.

In this work, we explore a highly-parallel and PIM-friendly implementation of deBruijn-based genome assembly that can accelerate genome assembly task. Overall this paper makes the following contributions: (1) To the best of our knowledge, this work is the first that designs a high-throughput X(N)OR-friendly PIM architecture exploiting DRAM arrays. We develop *PIM-Assembler* based on a set of novel microarchitectural and circuit-level schemes to realize a data-parallel computational unit for genome assembly; (2) We reconstruct the existing genome assembly algorithm such that it can be implemented in PIM 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 various length DNA sequences. (4) We extensively assess and compare *PIM-Assembler*'s performance and energy-efficiency with GPU and recent PIM platforms.

II. PIM-Assembler Architecture

A. Architecture

PIM-Assembler is designed to be an independent, high-performance, energy-efficient accelerator based on main memory architecture to accelerate a wide variety of applications.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2021 at 03:00:14 UTC from IEEE Xplore. Restrictions apply.

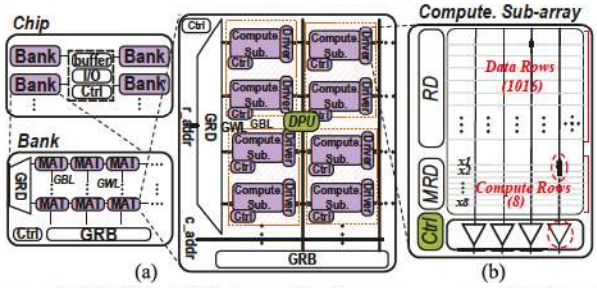

Figure 1: (a) The *PIM-Assembler*'s memory organization and contro, (b) Block level scheme of computational sub-array.

The main memory organization of *PIM-Assembler* is shown in Fig. 1a based on typical DRAM hierarchy. Each memory matrix (MAT) consists of multiple computational sub-arrays connected to a Global Row Decoder (GRD) and a shared Global Row Buffer (GRB). According to the physical address of operands within memory, *PIM-Assembler*'s Controller (Ctrl) is able to configure the sub-arrays to perform data-parallel intra-sub-array computations. A low-overhead Digital Processing Unit (DPU) is also considered in MAT-level to perform simple non-bulk bit-wise operations, as will be discussed later. We divide the *PIM-Assembler*'s sub-array row space into two distinct regions as depicted in Fig. 1b: 1- Data rows (1016 rows out of 1024) connected to a regular Row Decoder (RD), and 2- Computation rows (8-labeled by $x1, ..., x8$), connected to a Modified Row Decoder (MRD), which enables multi-row activation required for bulk bit-wise in-memory operations between operands. *PIM-Assembler*'s computational sub-array is developed to perform XNOR and addition operations leveraging charge-sharing among different rows.

With a careful observation on the existing processing-in-DRAM platforms, we realized that they are dealing with different challenges such as *Low Reliability* due to three/five row activation mechanisms [5], [6], *Row Initialization* [5], and *Extremely-Low Throughput X(N)OR* [3], [5], [6], which could be alleviated by rethinking about Sense Amplifier (SA) circuit. Our key idea is to perform in-memory logic operations through a new two-row activation mechanism. To achieve this goal, we propose a new reconfigurable SA, as shown in Fig. 2a, developed on top of existing DRAM circuitry. It consists of a regular DRAM SA equipped with add-on circuits including two inverters, one AND gate, one XOR gate, a D-latch, and one 4:1 MUX, controlled with five enable signals ($En_m, En_x, En_{mux}, En_{c1}, En_{c2}$). This design leverages the basic charge-sharing feature of DRAM cell and elevates it to implement XNOR2 and addition functions between the selected rows through static capacitive functions in one and two cycles, receptively. To implement capacitor-based logic, we use two different inverters with shifted Voltage Transfer Characteristic (VTC), as shown in Fig. 2b. In this way, a NAND/NOR logic can be readily carried out based on high switching voltage (V_s)/low-V_s inverters with standard high-V_{th}/low-V_{th} NMOS and low-V_{th}/high-V_{th} PMOS transistors. It is worth mentioning that utilizing low/high-threshold voltage transistors along with normal-threshold transistors have been

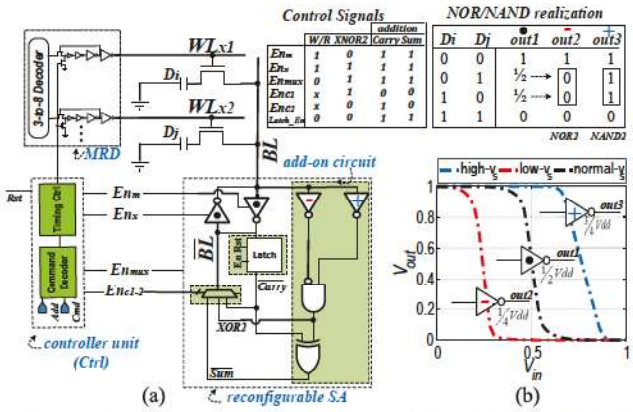

Figure 2: (a) New sense amplifier design for *PIM-Assembler*, (b) VTC and truth table of the SA's inverters.

accomplished in low-power application, and many circuits have enjoyed this technique in low-power design [14]. Fig. 2a shows the detailed control signals of *PIM-Assembler*'s sub-array to implement different memory and in-memory logic functions. *PIM-Assembler*'s ctrl activates En_m and En_x control-bits simultaneously (when MUX is deactivated-En_{mux}=0) to perform typical memory write/read operation. In memory operations, MUX's output voltage is high-z and $\overline{BL}$ voltage is solely determined in sense amplification state through two normal-V_s back-to-back inverters, just like normal DRAM's SA mechanism.

Now, considering D_i and D_j operands (in Fig. 2a) are copied (RowCloned [15]) from data rows to $x1$ and $x2$ computational rows, and both BL and $\overline{BL}$ are precharged to $\frac{V_{dd}}{2}$ (Precharged State). *PIM-Assembler*'s ctrl first activates two WLs in computational row space (here, $x1$ and $x2$) through the modified decoder for charge-sharing when all the other enable signals are deactivated. During sense amplification state, by setting the proper enable set (01110 for XNOR2), the input voltage of both low- and high-V_s inverters in the reconfigurable SA can be simply derived as $V_i = \frac{n.V_{dd}}{C}$, where n is the number of DRAM cells storing logic '1' and C represents the total number of unit capacitors connected to the inverters (i.e. 2 in our mechanism). Now, the low-V_s inverter acts as a threshold detector by amplifying deviation from $\frac{1}{4}V_{dd}$ and realizes a NOR2 function, as tabulated in the truth table in Fig. 2b. At the same time, the high-V_s inverter amplifies the deviation from $\frac{3}{4}V_{dd}$ and realizes a NAND2 function. Accordingly, we added a CMOS AND gate with one inverted input to realize XOR2 function between D_i and D_j. Now, SA's MUX can be readily reconfigured through the selectors to assign XOR2 value and its complementary logic to $\overline{BL}$ and BL, respectively. As can be seen, XNOR2 result can be produced in a single cycle on the BL. To realize addition operation, *PIM-Assembler* supports Ambit's Triple Row Activation (TRA) mechanism [5] to generate $Carry$ logic in a single cycle, where the result of this operation is stored in the SA's latch. The Ambit implements 3-input majority-based operations in memory by issuing the `ACTIVATE` command to three rows simultaneously followed by a single `PRECHARGE`

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2021 at 03:00:14 UTC from IEEE Xplore. Restrictions apply.

command. By activating the latch enable, the add-on XOR gate can generate $\overline{Sum}$ output in one cycle between two new input data and $Carry$ from previous cycle. The result is then inverted on the BL (Sum) and written back to the memory.

B. Performance Assessment

To assess the performance of *PIM-Assembler* as a new PIM platform, a comprehensive circuit-architecture evaluation framework and two in-house simulators are developed. 1- At the circuit level, we developed *PIM-Assembler*'s sub-array with new peripheral circuity (SA, MRD, etc.) in Cadence Spectre with 45nm NCSU Product Development Kit (PDK) library [16] to verify the two-row activation mechanism and achieve the performance parameters. 2- An architectural-level simulator is built on top of Cacti [17]. The circuit level results were then fed into our simulator. It can change the configuration files corresponding to different array organization and report performance metrics for PIM operations. The memory controller circuits are designed and synthesized by Design Compiler with a 45nm industry library. 3- A behavioral-level simulator is developed in Matlab to calculate the latency and energy that *PIM-Assembler* spends on different tasks. Besides, it has a mapping optimization framework to maximize the performance according to the available resources. The transient voltage simulation results of our PIM mechanism to realize single-cycle in-memory XNOR2 operation is shown in Fig. 3a. In this case, MUX's selectors are configured to set $\overline{BL}$ voltage with XOR2 result. We can see that cell's capacitor is accordingly charged to V_{dd} when D_iD_j=00/11 or discharged to GND when D_iD_j=10/01 during sense amplification state.

- **Throughput:** We evaluate and compare the *PIM-Assembler*'s raw performance with different computing units and accelerators including a Core-i7 Intel CPU [18] and an NVIDIA GTX 1080Ti Pascal GPU. In PIM domain, we shall restrict our comparison to four recent processing-in-DRAM platforms, Ambit [5], DRISA-1T1C (D1) [3], DRISA-3T1C (D3) [3], and HMC 2.0 [19], to handle two operations. To have a fair comparison, we report *PIM-Assembler*'s and other PIM platforms' raw throughput implemented with 8 banks with 1024×256 computational sub-arrays. The Intel

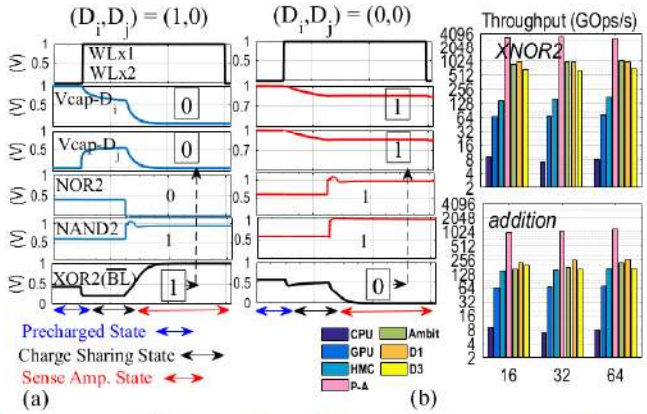

Figure 3: (a) The transient simulation of in-memory XNOR2 operation, (b) Throughput of XNOR2 and addition operations implemented by different platforms.

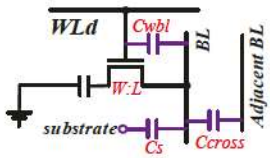

Figure 4: Noise sources in DRAM cell. Glossary: $Cwbl$, Cs, and $Ccross$ are WL-BL, BL-substrate, and BL-BL capacitance, respectively.

Variation	TRA	2-Row act.
±5%	0.00	0.00
±10%	0.18	0.00
±15%	5.5	1.6
±20%	17.1	11.2
±30%	28.4	18.1

Table I: Process variation.

CPU consists of 4 cores and 8 threads working with two 64-bit DDR4-1866/2133 channels. The Pascal GPU has 3584 CUDA cores running at 1.5GHz and 352-bit GDDR5X. The HMC has 32-10 GB/s bandwidth vaults. Accordingly, we develop an in-house micro-benchmark to run the operations repeatedly for $2^{27}/2^{28}/2^{29}$-bit length input vectors and report the throughput of each platform, as shown in Fig. 3b. We observe that either the external or internal DRAM bandwidth has limited the throughput of the CPU, GPU, and even HMC platforms. Besides, our platform (indicated by P-A) improves the throughput on average by 2.3×, 1.9×, 3.7× compared with Ambit [5], D1 [3], and D3 [3], respectively. This mainly comes from the single cycle X(N)OR mechanism that eliminates the need for row the initialization in Ambit and multi-cycle DRISA mechanism.

- **Software Support:** *PIM-Assembler* is developed based on `ACTIVATE-ACTIVATE-PRECHARGE` command a.k.a. `AAP` primitives and most bulk bitwise operations involve a sequence of `AAP` commands. To enable processor to efficiently communicate with *PIM-Assembler*, we developed three types of `AAP`-based instructions that only differ from the number of activated source or destination rows: 1- `AAP (src, des, size)` that runs the following commands sequence: 1) `ACTIVATE` a source address (`src`); 2) `ACTIVATE` a destination address (`des`); 3) `PRECHARGE` to prepare the array for the next access. The `size` of input vectors for in-memory computation must be a multiple of DRAM row size, otherwise the application must pad it with dummy data. The type-1 instruction is mainly used for copy function; 2- `AAP (src1, src2, des, size)` that performs two-row activation method by activating two source addresses and then writes back the result on a destination address; 3- `AAP (src1, src2, src3, des, size)` that performs Ambit-TRA mechanism [5] by activating three source rows and writing back the result on a destination address.

- **Reliability:** We performed a comprehensive circuit-level simulation to study the effect of process variation on both two-row activation and TRA methods considering different noise sources and variation in all components including DRAM cell (BL/WL capacitance and transistor, shown in Fig. 4) and SA (width/length of transistors-V_s). We ran Monte-Carlo simulation in Cadence Spectre (DRAM cell parameters were taken and scaled from Rambus [20]) under 10000 trials and increased the amount of variation from ±0% to ±30% for each method. Table I shows the percentage of the test error in

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2021 at 03:00:14 UTC from IEEE Xplore. Restrictions apply.

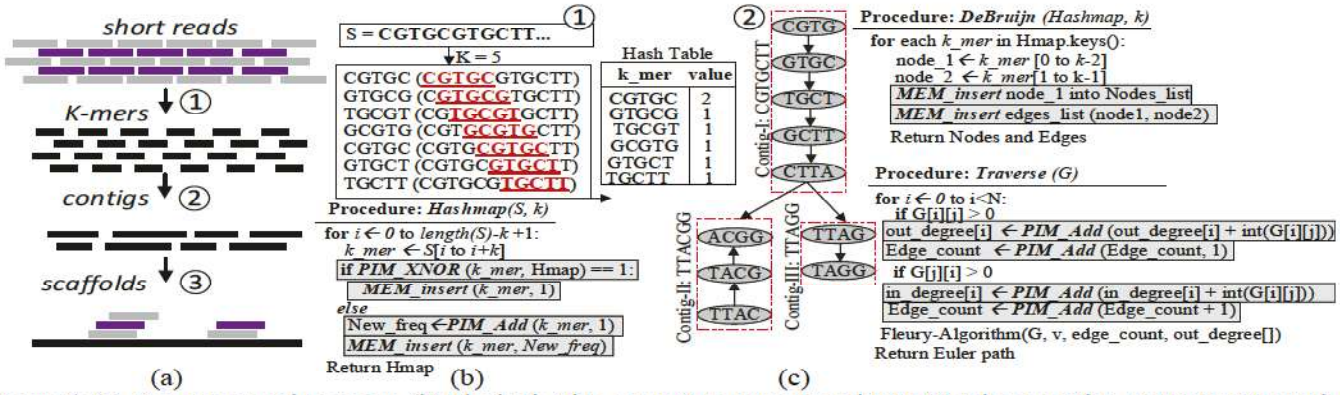

Figure 5: (a) Genome assembly stages, (b) The hash table generation example and PIM-friendly algorithm, (c) DeBruijn graph construction and traversing stage for contig. generation stage.

each variation. We observe that even considering a significant $\pm 10\%$ variation, the percentage of erroneous 2-row activation mechanism across 10000 trials is *zero*, where TRA method shows a failure with 0.18%. By scaling down the transistor size, the process variation effect is expected to get worse [5], [15]. Since *PIM-Assembler* is mainly developed based on existing DRAM structure and operation with slight modifications, different methods currently-used to tackle process variation can be also applied for *PIM-Assembler*. Therefore, *PIM-Assembler* offers a solution to alleviate different bottlenecks in current processing-in-DRAM designs.

• **Area Overhead:** To estimate the area overhead of *PIM-Assembler* on top of commodity DRAM chip, three hardware cost sources must be taken into consideration. First, add-on transistors to SAs; in our design, each SA requires ~50 additional transistors connected to each BL. Second, the 3:8 MRD overhead; we modify each WL driver by adding two more transistors in the typical buffer chain, as depicted in Fig. 2a, so there is only 16 add-on transistors for computational rows. Third, the Ctrl's overhead to control enable bits. To sum it up, *PIM-Assembler* imposes 51 DRAM rows (51×256 transistors) per sub-array, at the most, which can be interpreted as $\sim 5\%$ of DRAM chip area.

III. PIM-Assembler Algorithm and Mapping

The genome assembly algorithm consists of three stages visualized in Fig. 5a, first, *k*-mer analysis to create a hashmap (hash table) out of chopped short reads (*k*-mers), second, contig generation to construct deBruijn graph with hashmap and traverse through it for Euler path and third, scaffolding to close the gaps between contigs, which is the result of the denovo assembly [2]. The first two stages always take most fraction of execute time and computational resources (over 80%) in both CPU and GPU implementations [2]. Therefore, we mainly focus on parallelizing these steps using *PIM-Assembler*'s functions, and leave stage-3 as our future work.

• ***k*-mer analysis:** Fig. 5b shows the reconstructed *Hashmap(S,k)* procedure in which the algorithm takes *k*-mer from original sequence (*S*) in each iteration, creates a hash table entry for that, and assigns its frequency (value) to 1. If the *k*-mer is already in the table, it will calculate a new frequency (New_freq) by adding the previous frequency by one and update the value. As shown, hashmap procedure can be reconstructed through *PIM_XNOR* (comparison), *PIM_Add* (addition), and *MEM_insert* (memory W/R operation) in-memory functions. Such functions are iteratively used in every step of 'for' loop and *PIM-Assembler* is specially designed to handle such computation-intensive load through performing comparison, summing, and copying operations. Due to large memory space requirement for hash table for assembly-in-memory algorithm, we partition these tables to multiple sub-arrays to fully leverage *PIM-Assembler*'s parallelism, and to maximize computation throughput. The proposed correlated data partitioning and mapping methodology, as shown in Fig. 6, 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. For counting 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 shown in Fig. 6, 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 can be performed between temp data and already-stored *k*-mers. Fig. 7 intuitively shows *PIM_XNOR* procedure, where entire temp row can be compared with a previous *k*-mer row in a single cycle. Then, a built-in AND unit in DPU readily takes all the results to determine the next memory operation according to the algorithm.

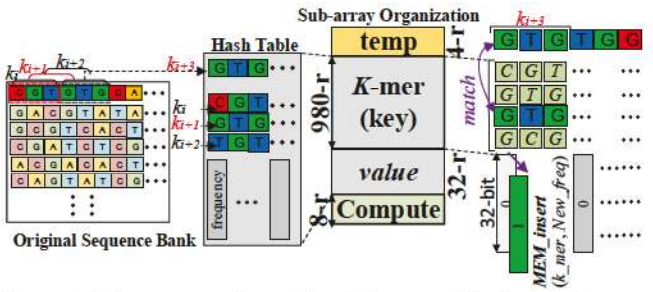

Figure 6: The proposed correlated data partitioning and mapping methodology for creating hash table.

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2021 at 03:00:14 UTC from IEEE Xplore. Restrictions apply.

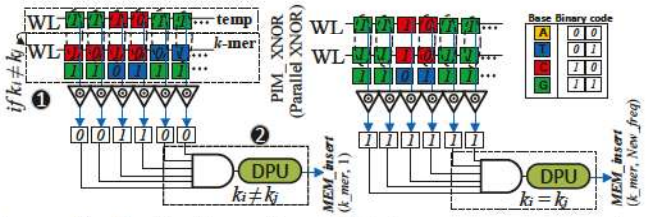

Figure 7: Realization of parallel in-memory comparator (*PIM_XNOR*) in *PIM-Assembler*'s computational sub-array.

• **Contig generation:** For graph construction and traversal, we adopt interval-block partitioning method to balance workloads of each *PIM-Assembler*'s chip and maximize parallelism. We utilize hash-based method [21], [22] to divide the vertices into M intervals and then divide edges into M^2 blocks as Fig. 8 (partitioning stage). Then each block is allocated to a chip and mapped to its sub-arrays. Having an N-vertex sub-graph with N_s activated sub-arrays (size=$a \times b$), each sub-array can process n vertices ($n \leq f | n \in N, f = min(a, b)$). So, the number of sub-arrays for processing an N-vertex sub-graph can be formulated as, $N_s = \left\lceil \frac{N}{f} \right\rceil$ (allocation stage). In this stage, the designated graph is converted to an adjacency matrix and mapped to consecutive rows of *PIM-Assembler*'s sub-arrays. The reconstructed *DeBruijn (Hashmap,k)* and *Traverse (G)* procedures in Fig. 5 deal with massive number of iteratively-used *MEM_insert* and *PIM_Add* operations. In the interest of space, we focus on out/in degree calculation in traverse procedure, which basically sums up all the entries of a particular node i of valid links connected to a vertex to find the start vertex. Fig. 8 shows an intuitive example of hardware mapping and acceleration of such operation performed by *PIM-Assembler* for a sample graph converted to adjacency matrix and mapped to consecutive rows of *PIM-Assembler*'s sub-arrays. To perform parallel addition operation and generate initial Carry (C) and Sum (S) bits (mapping stage), *PIM-Assembler* takes every three rows to perform a parallel in-memory addition. The results are written back to the memory reserved space (Resv.). Then, next step only deals with multi-bit addition of resultant data starting bit-by-bit from the LSBs of the two words continuing towards MSBs. This process concluded after $2 \times m$ cycles, where m is number of bits in elements. At the end the degree of each vertex is stored in memory (e.g., in Fig. 8, 4 determines the degree of vertex 1).

IV. Performance Estimation

• **Setup:** To the best of our knowledge, this work is the first to discuss the PIM platform's performance for genome assembly problem, therefore, we create the evaluation test bed from scratch. We configure the *PIM-Assembler*'s memory sub-array with 1024 rows and 256 columns, 4×4 mats (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 group. For comparison with other PIM platforms, an identical physical memory configuration is also considered henceforth. We conduct our experiment on human chromosome-14 data-set. We create the short reads

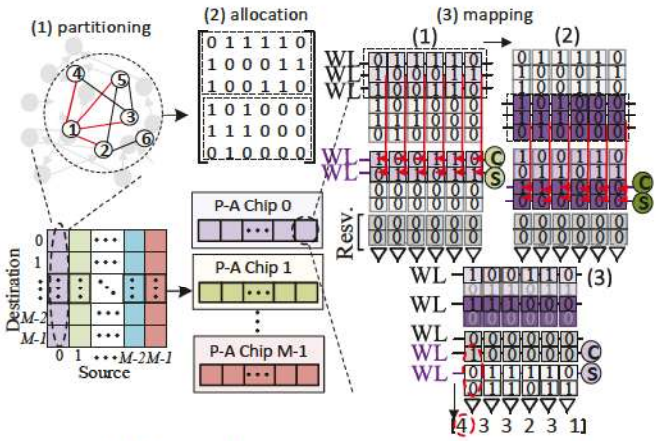

Figure 8: *PIM-Assembler*'s partitioning, allocation and mapping to accelerate *PIM_Add* operation.

(45,711,162) with the length of 101 (total memory requirement ∼9.2GB), by randomly sampling the chromosome extracted from the NCBI genome databases [23]. We set the k-mer length, k, to 16, 22, 26, and 32, as typical values for most genome assemblers in order to run and estimate the performance of three main procedures in genome assembly show in Fig. 5 i.e. hashmap, deBruijn, and traverse.

• **Execution Time:** Fig. 9a reports and compares the execution time of *PIM-Assembler* with the under-test GPU platform used in Section II.B and other processing-in-DRAM platforms including Ambit [5], DRISA-3T1C and DRISA-1T1C [3]. As shown, hashmap procedure for k-mer analysis takes the largest fraction of execution time and power in GPU platform (over 60%). We observe that our X(N)OR-friendly platform can accelerate the hashmap generation by ∼5.2× compared with GPU platform when k=16. Now, by increasing the k-length to 32, the higher speed-up is even achievable (∼9.8×). As for *PIM_Add*, and *MEM_insert* in-memory functions used in deBrujin and traverse procedures, we can see *PIM-Assembler* outperforms the GPU platform by 4.2× higher performance. Overall, we observe that *PIM-Assembler* reduces the execution time on average by 5×, 2.9×, 2.5× and 2.8× as compared to GPU, Ambit [5], DRISA-3T1C and DRISA-1T1C [3] platforms, respectively.

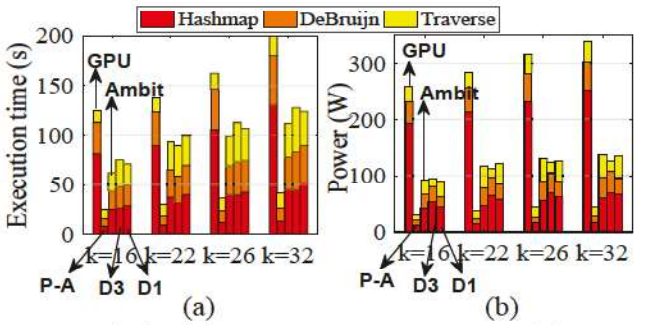

Figure 9: The breakdown of (a) Execution time and (b) Power consumption for different platforms running different k-mer-length genome assembly task. In each bar group from left to right: GPU, *PIM-Assembler* (indicated as P-A), Ambit, DRISA-3T1C (D3), DRISA-1T1C (D1).

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2021 at 03:00:14 UTC from IEEE Xplore. Restrictions apply.

• **Power:** We report the estimated power consumption of different platforms for running different length k-mers. Based on Fig. 9b, *PIM-Assembler* shows the least power consumption (on average 38.4W) to run the three main procedures, as compared with the GPU and other PIM platforms. The *PIM-Assembler* reduces the power consumption by ∼7.5× compared with the GPU platform. Besides, it achieves ∼2.8× lower power vs. the best PIM platform.

• **Trade-offs:** We explore the efficiency of the platform by adjusting the number of active sub-arrays (N_s) in processing the *PIM_XNOR* and *PIM_Add* functions. We define a parallelism degree (P_d) i.e. the number of replicated sub-arrays to increase the performance, e.g., when P_d is set to 2, we use two parallel sub-arrays to simultaneously process the functions. It turns out such parallelism enhances the performance of genome assembly by sacrificing the chip area and power consumption. Based on this, we plotted Fig. 10 to show the trade-off between power and delay vs. P_d for two distinct k-mer lengths i.e. 16 and 32. It can be seen that the larger P_d is, the smaller delay and higher power consumption are resulted for both configurations. Therefore, we determine the optimum performance of *PIM-Assembler*, where $P_d \backsimeq 2$.

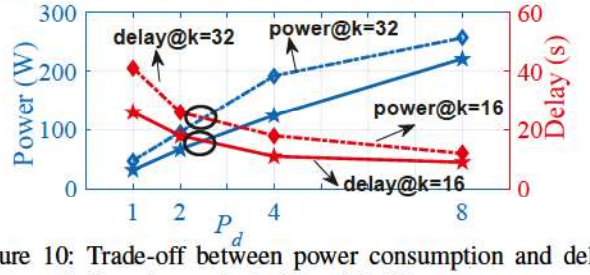

Figure 10: Trade-off between power consumption and delay w.r.t parallelism degree in k=16 and k=32.

• **Memory Wall:** Fig. 11a reports the Memory Bottleneck Ratio (MBR), as the time that the computation waits for data and on-/off-chip data transfer blocks the performance. We conduct the evaluation w.r.t. the peak throughput for each platform in two distinct k-mer lengths considering number of memory access. The results indicate the *PIM-Assembler* and other PIMs' efficiency for solving memory wall issue. We see that *PIM-Assembler* uses less than ∼16% time for data transfer due to the PIM acceleration schemes, while GPU's MBR increases to 70% when k=32. The smaller MBR can be translated as the higher Resource Utilization Ratio (RUR) for the accelerators plotted in Fig. 11b. We observe *PIM-Assembler* has the highest RUR with up to ∼65% when k=16. Overall, PIM solutions give a higher ratio (>45%) compared to the GPU authenticating the results in Fig. 11a.

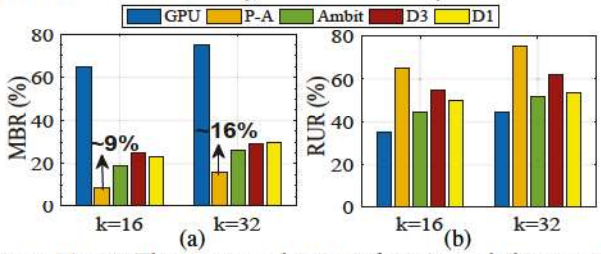

Figure 11: (a) The memory bottleneck ratio and (b) resource utilization ratio.

V. Conclusion

In this work, we presented *PIM-Assembler*, as a new PIM architecture to address some of the existing issues in state-of-the-art DRAM-based acceleration solutions for performing bulk bit-wise X(N)OR-based operations. Accordingly, we show how *PIM-Assembler* can accelerate the comparison/addition-extensive genome assembly application using PIM-friendly operations. The simulation results on human-ch14 shows that this new platform reduces the execution time and power by ∼5× and ∼7.5× compared to GPU.

VI. Acknowledgement

This work is supported in part by the National Science Foundation under Grant No.2005209, No.2003749, No. 1755761 and Semiconductor Research Corporation nCORE.

References

[1] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[2] E. Georganas *et al.*, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 437–448.

[3] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Micro*. ACM, 2017, pp. 288–301.

[4] S. Angizi *et al.*, "Aligns: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram," in *DAC*, 2019, p. 144.

[5] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Micro*. ACM, 2017, pp. 273–287.

[6] S. Angizi and D. Fan, "Graphide: A graph processing accelerator leveraging in-dram-computing," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. ACM, 2019, pp. 45–50.

[7] R. Li *et al.*, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, pp. 1966–1967, 2009.

[8] C.-M. Liu *et al.*, "Soap3: ultra-fast gpu-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, pp. 878–879, 2012.

[9] J. Arram *et al.*, "Leveraging fpgas for accelerating short read alignment," *IEEE/ACM TCBB*, vol. 14, pp. 668–677, 2017.

[10] S. F. Mahmood and H. Rangwala, "Gpu-euler: Sequence assembly using gpgpu," in *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE, 2011, pp. 153–160.

[11] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, pp. 821–829, 2008.

[12] M. Lu *et al.*, "Gpu-accelerated bidirected de bruijn graph construction for genome assembly," in *Asia-Pacific Web Conference*, 2013, pp. 51–62.

[13] S. Angizi and D. Fan, "Redram: A reconfigurable processing-in-dram platform for accelerating bulk bit-wise operations," in *ICCAD*. IEEE, 2019, pp. 1–8.

[14] M. W. Allam *et al.*, "High-speed dynamic logic styles for scaled-down cmos and mtcmos technologies," in *ISLPD*. ACM, 2000, pp. 155–160.

[15] V. Seshadri *et al.*, "Rowclone: fast and energy-efficient in-dram bulk data copy and initialization," in *Micro*. ACM, 2013, pp. 185–197.

[16] (2011) Ncsu eda freepdk45. [Online]. Available: http://www.eda.ncsu.edu/wiki/FreePDK45:Contents

[17] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[18] "6th generation intel core processor family datasheet." [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/core/core-vpro/i7-6700.html

[19] "Hybrid memory cube speci!cation 2.0." [Online]. Available: http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.0_Public.pdf.

[20] . DRAM Power Model. https://www.rambus.com/energy/.

[21] G. Dai *et al.*, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE TCAD*, 2018.

[22] S. Angizi *et al.*, "Graphs: A graph processing accelerator leveraging sot-mram," in *DATE*. IEEE, 2019, pp. 378–383.

[23] "National center for biotechnology information." [Online]. Available: http://www.ncbi.nlm.nih.gov/

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2021 at 03:00:14 UTC from IEEE Xplore. Restrictions apply.