

LAMP: An Adaptive Near-Memory Processing System for High-Performance Long-Read Mapping

Jo-Ling Huang

Department of CSIE

National Cheng Kung University

Tainan, Taiwan (R.O.C.)

p76124841@gs.ncku.edu.tw

Liang-Chi Chen

Department of CSIE

National Taiwan University

Taipei, Taiwan (R.O.C.)

d12922012@csie.ntu.edu.tw

Chien-Chung Ho

Department of CSIE

National Cheng Kung University

Tainan, Taiwan (R.O.C.)

ccho@gs.ncku.edu.tw

Yuan-Hao Chang

Department of CSIE

National Taiwan University

Taipei, Taiwan (R.O.C.)

johnson@csie.ntu.edu.tw

Abstract—Long-read sequencing technologies have improved genome assembly and structural variant detection, but impose heavy computational demands on alignment tools, which remain bottlenecked by memory-bound seeding and chaining phases. This work presents LAMP, an adaptive near-memory processing (NMP) framework that coordinates CPU execution with the NMP platform to accelerate these stages. LAMP introduces two key mechanisms: (1) a load-spreading hashing scheme that balances minimizer distribution for processing units during index construction, and (2) a density-aware adaptive dispatching strategy that mitigates workload skew in chaining by partitioning reads according to anchor density. In the experiments, LAMP achieves up to $3.8\times$ speedup over CPU-only execution. The results show that system-level co-design can overcome the memory bandwidth and load imbalance challenges of NMP architectures, enabling scalable, high-throughput genomic analysis.

I. INTRODUCTION

Recent advances in third-generation sequencing technologies, such as Oxford Nanopore and PacBio, have enabled the generation of ultra-long DNA reads, often spanning tens of kilobases. These long reads have significantly improved genomic analysis, including de novo genome assembly, structural variant detection, and the resolution of complex or repetitive genomic regions. These capabilities are often difficult to achieve with traditional short-read sequencing technologies. However, these benefits come with increased computational demands. Long-read sequencing technologies tend to produce reads with higher error rates, typically ranging from 10% to 15% per base, meaning that one in every 7 to 10 bases may be incorrect. In combination with the large volume of data produced, this significantly increases the complexity and cost of sequence alignment. To address these challenges, state-of-the-art long-read aligners such as Minimap2 [1], NGMLR [2], and Winnowmap [3] adopt a seed-chain-align paradigm. This approach narrows the alignment search space by first identifying short matching segments (seeds), then assembling them into longer candidate chains, and finally applying base-level alignment within promising regions. While this design improves throughput and reduces unnecessary computation, the mapping process remains dominated by memory-bound operations, which place substantial stress on conventional CPU architectures.

At the same time, the growing gap between processor speed and memory bandwidth, commonly referred to as the von Neumann bottleneck, has become a significant obstacle for data-intensive workloads. Although accelerators such as GPUs and FPGAs can offer meaningful speedups, their benefits are

often limited by the cost of frequent data movement between compute units and memory, especially in memory-bound tasks like alignment. To overcome these limitations, near-memory processing (NMP) architectures have emerged as a promising alternative. By placing compute units in close physical proximity to memory, NMP systems significantly reduce data transfer overhead and improve both memory bandwidth utilization and energy efficiency. Among these, the UPMEM platform [4] offers a commercially available solution that integrates thousands of lightweight processing units directly into standard DRAM DIMMs, enabling massive parallelism and high aggregate bandwidth. Despite these advantages, directly offloading long-read mapping tasks to NMP platforms introduces several architectural challenges. Specifically, the *limited local memory per processing unit*, *rigid control granularity at the rank level*, and *data-dependent workload skew* hinder naïve offloading strategies. For example, index construction may exceed the local memory capacity of individual units. Chaining operations often suffer from severe imbalance due to uneven anchor distributions. In addition, host-device data transfers can become bottlenecks if they are not carefully coordinated. These challenges motivate the need for a system-aware design to fully exploit the potential of NMP architectures in genomic workloads.

To address these challenges, we propose LAMP (long-read mapping with an adaptive NMP system), a near-memory computing framework designed to accelerate long-read alignment on real NMP platforms. The core contribution of LAMP lies in its adaptive management strategy, which mitigates workload imbalance caused by heterogeneous read lengths and architectural constraints of NMP systems. It also balances memory usage across processing units and reduces communication overhead between the host and the device. Our results show that, with careful system-level co-design, NMP architectures can be effectively used to accelerate memory-bound stages such as index construction and chaining. This approach avoids the inefficiencies of naïve task offloading and provides a scalable solution for long-read mapping.

II. BACKGROUND

A. Long-read Mapping

Modern long-read mappers typically adopt a *seeding-chaining-alignment* paradigm to efficiently process high-error, high-throughput sequencing data. To illustrate this workflow, we

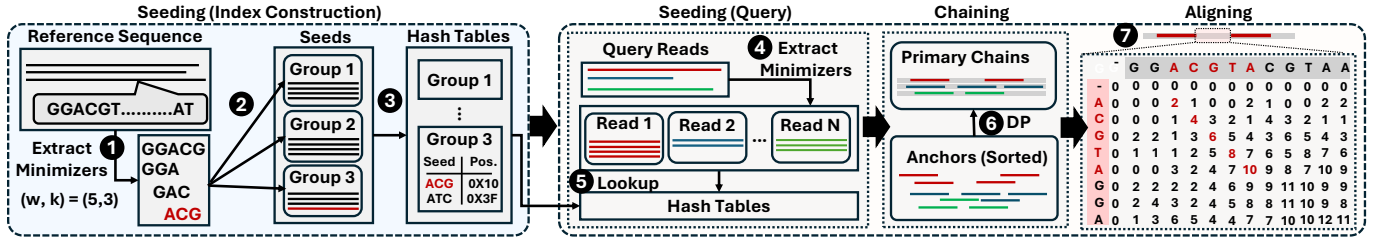


Fig. 1. Workflow of long-read mapping.

describe the procedure of Minimap2 [1], one of the most widely used aligners for long-read data, as shown in Figure 1.

In the seeding stage, Minimap2 builds an index from the reference genome using a minimizer-based sampling method [5]. It slides a window of size w across the reference to extract overlapping k -mers, and selects the one with the smallest hash value as the *minimizer*, which serves as a representative seed. Seeds are grouped by hash prefix into buckets and stored in a hash-indexed lookup table along with their positions. This sparse sampling reduces redundancy while maintaining sensitivity. After the reference index is constructed, Minimap2 processes each query read in a similar manner. Minimizers are extracted from the read using the same windowed hashing procedure, and matching minimizers between the query and the reference are identified via the hash table. Each matched minimizer, termed an anchor, indicates a potential region of homology between the read and the reference, and serves as the input to the chaining phase.

In the chaining stage, Minimap2 performs one-dimensional dynamic programming (DP) over the sorted list of anchors to identify high-scoring collinear subsets, known as chains. Each anchor is represented as a 3-tuple: (r, q, l) , where r is the reference coordinate, q is the query coordinate, and l is the length of the match. Anchors are first sorted by their reference positions. For each anchor, the algorithm searches a dynamically bounded set of predecessors to compute the optimal chain score. This score is computed based on gap consistency (between adjacent anchors), overlap length, and strand orientation. The chain with the highest score is selected as the primary chain, while other overlapping but lower-scoring chains are designated as secondary chains. This phase significantly narrows the search space for the next step by focusing only on a small number of promising alignment regions. In the final alignment stage, Minimap2 applies base-level alignment to regions between anchors within the primary chain. Using Smith–Waterman–Gotoh with affine gap penalties [6], this step refines the alignment to capture substitutions, insertions, and deletions, producing the final base-to-base alignment and mapping score.

B. Near-Memory Processing (NMP) systems

System Architecture. Near-memory processing (NMP) has emerged as a promising approach to address memory bottlenecks that constrain conventional CPU-centric architectures. By placing lightweight processing units adjacent to the memory, NMP systems improve memory bandwidth utilization and energy efficiency by minimizing data movement. Several com-

mercial and academic NMP architectures have been proposed, including Samsung’s AXDIMM [7], SK Hynix’s AiM [8], and UPMEM’s DPU system [4]. These designs integrate small compute cores near or within DRAM chips to accelerate data-intensive workloads. Among them, UPMEM represents one of the most mature and widely studied NMP platforms [9]–[16], and is adopted in this study as the reference NMP architecture.

As illustrated in Figure 2, a typical UPMEM system includes both standard and NMP-enabled DIMMs. While standard DIMMs serve as host main memory, NMP DIMMs function as accelerators and contain one or two ranks, each consisting of eight PIM-enabled DRAM chips. Each chip integrates eight processing units (PUs), yielding 64 PUs per rank. These PUs operate independently and concurrently. Each PU comprises a 32-bit multithreaded RISC core with 24 KB instruction memory, 64 KB scratchpad memory for fast local computation, and a private 64 MB DRAM bank. A DMA engine coordinates data transfers among the core, scratchpad, and DRAM bank.

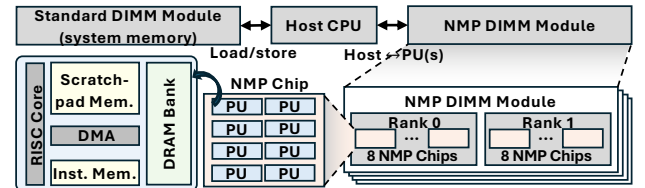


Fig. 2. A typical near-memory processing system [17].

Program execution on NMP systems typically follows a four-stage accelerator model: (1) The host compiles and loads PU instructions; (2) input data is explicitly transferred from host memory to PU-local DRAM¹; (3) the host launches PU execution; and (4) results are transferred back to the host upon completion.

Constraints and Limitations in NMP Systems. While NMP provides substantial potential for memory bandwidth and parallelism, its architectures impose constraints that complicate scalable application design. (1) Each PU is restricted to a fixed-size DRAM bank, which limits the data volume that can be processed at once and requires careful workload partitioning. Since inter-PU communication is not supported, any data exchange must pass through the host, resulting in additional overhead. (2) DDR-based DIMM structures enforce control granularity at the rank level, meaning all PUs in a rank must finish their tasks before subsequent data transfer or computation can begin. This

¹We use “host-PUs” to denote host-to-PUs transfers and “PUs-host” to denote PUs-to-host transfers.

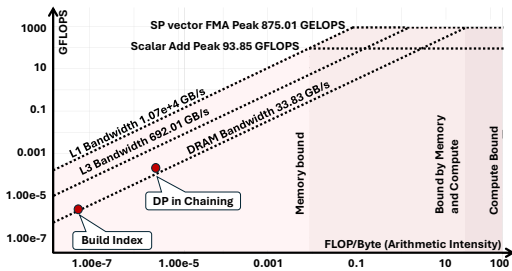


Fig. 3. Roofline analysis of minimap2.

amplifies the effects of workload imbalance, as slower PUs can stall an entire rank. (3) Data transfers between the host and PUs cannot overlap with PU computation due to software-level and hardware-level constraints [20], [21], necessitating careful scheduling to avoid idle periods. In summary, although NMP systems provide high aggregate memory bandwidth and extensive parallelism, they require meticulous data distribution and load balancing to achieve efficient execution.

C. Motivation

Profiling Long-Read Mapping. To identify the performance bottlenecks in long-read mapping, we profile Minimap2 [1], a widely used long-read aligner. Despite its high accuracy and throughput on modern CPUs, Minimap2 remains limited by memory bandwidth, especially during memory-intensive stages. Using Intel Advisor [22], we perform a roofline analysis on a conventional CPU platform [23]. As shown in Figure 3, two stages, i.e., index construction and chaining, exhibit low arithmetic intensity and are clearly memory-bound. In the *index construction* stage, reference minimizers are inserted into hash buckets. This process involves frequent hashing and memory writes with limited data reuse, leading to over 60 GB of memory traffic. Poor spatial and temporal locality further reduces CPU cache effectiveness, and the large reference genome exacerbates hash table storage costs. The *chaining* stage applies dynamic programming over sorted anchors to form high-scoring collinear chains. Although the computation itself is simple, irregular memory access patterns and the large anchor volume result in poor cache behavior and bandwidth saturation. As read length and sequencing depth grow, chaining increasingly dominates the runtime.

Challenges of Indexing Stage. The *indexing stage*, which inserts extracted minimizers into hash tables, is naturally suited for NMP acceleration due to its high memory throughput demands. To leverage this, we implement a parallel indexing pipeline where the host partitions the reference and assigns groups of minimizers to PUs for local hash table construction. However, as shown in Figure 4, some minimizer groups are highly dense and may exceed the 64 MB DRAM capacity of a PU. Handling such cases requires either multi-pass processing or distribution across multiple PUs, both of which incur overhead from repeated host-PU transfers or indirect synchronization. Efficient workload partitioning under strict memory limits becomes a critical challenge [24].

Challenges of Chaining Stage. The *chaining stage* also presents significant challenges due to workload skew. Anchor

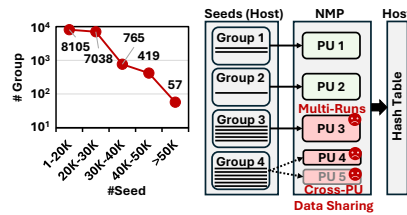


Fig. 4. Challenges of NMP-based index construction. The left part is the distribution of minimizer counts per group in the Hg38 [18].

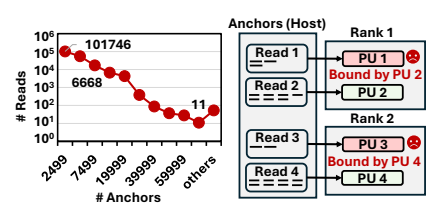


Fig. 5. Challenges of NMP-based chaining. The left part is the distribution of anchor counts per read in a real dataset [19].

density varies widely among reads, with a small fraction of long or repetitive reads accounting for a disproportionate share of anchors, as shown in Figure 5. Naively distributing reads across PUs leads to imbalanced workloads, where some PUs are heavily loaded while others remain underutilized. This imbalance is further amplified by the rank-level control granularity in NMP systems, where all PUs in a rank must complete execution before subsequent transfers or tasks can begin. Splitting anchor-heavy reads is often infeasible due to data dependencies within chaining, and anchor density cannot be predicted ahead of time, making static scheduling ineffective.

In summary, while NMP systems provide high memory bandwidth and substantial parallelism, they also introduce architectural challenges. Limitations such as constrained local memory, rank-level execution control, and data-dependent workload imbalance hinder efficient task offloading. These challenges motivate the development of an adaptive NMP-based framework for long-read mapping.

III. LAMP: LONG-READ MAPPING WITH AN ADAPTIVE NMP SYSTEM

A. Design Overview

We present LAMP (long-read mapping with an adaptive NMP system), a near-memory computing framework designed to accelerate long-read alignment on NMP systems. LAMP addresses two key challenges in memory-bound mapping pipelines: (1) irregular memory access patterns during large-scale reference indexing, and (2) dynamic programming (DP)-based chaining under highly skewed anchor distributions. To overcome these challenges, LAMP incorporates two main components: (1) a *load-spreading hashing mechanism* to balance minimizer distribution across PUs during index construction, and (2) a *density-aware adaptive dispatching strategy* to partition chaining workloads according to anchor density, improving parallel efficiency.

Overall Workflow. As shown in Figure 6, the LAMP workflow begins with reference preprocessing on the host. Minimizers are extracted and grouped using the *load-spreading hashing mechanism*, which ensures balanced bucket sizes for downstream parallelism. These minimizer groups are then distributed to PUs, each of which constructs a local hash table independently. The resulting tables are collected and stored on the host.

During query processing, the host extracts minimizers from incoming reads and uses the PU-generated hash tables to identify anchors. Anchors are dispatched using the *density-aware adaptive dispatching strategy*: anchor-dense reads are offloaded

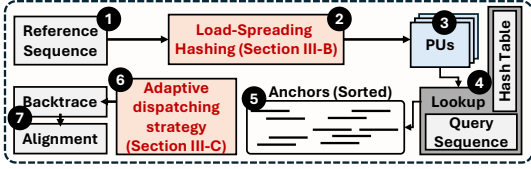


Fig. 6. Overall workflow of the proposed LAMP.

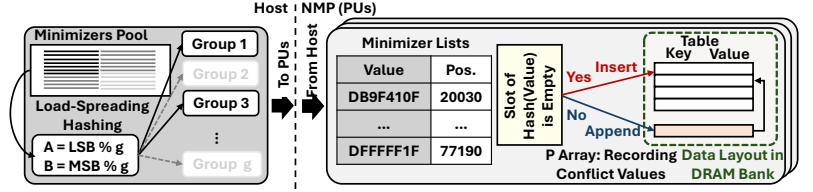


Fig. 7. Load-spreading hashing mechanism and minimizer insertion logic on PUs.

to PUs for parallel DP-based chaining, while anchor-sparse reads are processed in larger batches or retained for host-side execution. This design improves utilization of both compute domains by mitigating idle time from skewed anchor distributions. Finally, alignment refinement, including backtracking after DP, is performed on the host to preserve compatibility with existing mapping pipelines. Detailed designs for the hashing and dispatching mechanisms are described in Sections III-B and III-C, respectively.

B. Load-Spreading Hashing Mechanism

A key challenge in index construction is the uneven distribution of minimizers, which can cause workload imbalance and exceed the memory capacity of individual PUs. To mitigate this, we propose a *load-spreading hashing mechanism* that evenly distributes minimizers across PUs while preserving insertion locality and parallelism. This design leverages two key properties: (1) minimizer insertions are inherently independent, enabling massive parallelism across PUs; and (2) load distribution across PUs must be balanced to prevent memory overflow and maximize utilization. In our proposed load-spreading hashing mechanism, minimizers extracted from the reference genome are initially pooled on the host. Instead of using a naïve modulo-based group assignment, we apply a dual-hashing strategy that computes two candidate group IDs using deterministic bit-shifting operations: $(\text{value} \gg 8) \% \text{total_groups}$ and $(\text{value} \gg 16) \% \text{total_groups}$. This two-choice hashing technique is known to reduce maximum bucket load to $O(\log \log n)$ with high probability [25]. The host compares the current load of the two candidate groups and assigns each minimizer to the group with the lower count. This process reduces the likelihood of hotspots from high-frequency minimizers while maintaining deterministic lookup behavior and avoiding randomization overhead.

Once each minimizer is assigned a group ID, groups are evenly distributed across the PUs. Each PU independently inserts its assigned minimizers into a local key-value hash table. The PU first loads minimizers from its DRAM bank into its working memory. After computing hash values, it updates the corresponding entries in the hash table, which resides in the same DRAM bank. As shown in Figure 7, if a hash entry is empty, the PU inserts the minimizer key and its reference position directly. If the key already exists, the new position is appended to the existing position list P , which records all occurrences of the same minimizer. This localized design allows efficient handling of duplicates and enables compact storage with fast lookups. All insertions occur independently across PUs without the need for synchronization, supporting high-throughput, fine-grained parallelism during index construction.

C. Density-Aware Adaptive Dispatching Strategy

To mitigate workload imbalance caused by variation in anchor density across reads, we design a *density-aware adaptive dispatching* strategy that dynamically classifies reads and assigns computation to either the host or the PUs. The overall procedure is illustrated in Figure 8.

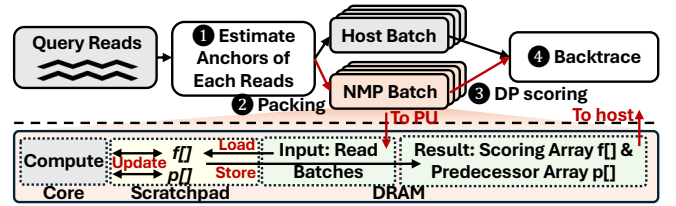


Fig. 8. The density-aware adaptive dispatching for the changing phase.

In step 1, the host estimates the anchor count for each read and classifies reads into two categories based on a predefined threshold. Reads with high anchor density (referred to as anchor-dense reads) are designated for offloading to the PUs, while those with low anchor density (referred to as anchor-sparse reads) are retained for host-side processing. In step 2, classified reads are organized into execution batches. Anchor-dense reads are grouped into fixed-size batches that align with the memory constraints and control granularity of the PUs. Anchor-sparse reads are batched for efficient parallel processing on the host, minimizing overhead. In step 3, each processing unit performs forward dynamic programming (DP) scoring over its assigned reads. The chaining logic follows a windowed recurrence: $f[i] = \max_{j < i} \{f[j] + w_{j,i}\}$, where $w_{j,i}$ represents the chaining weight based on distance penalties, overlap penalties, and colinearity constraints. Score array and predecessor array $f[]$, $p[]$ are allocated in local DRAM and scratchpad memory of each PU to ensure high memory locality and efficient inner-loop computation. Score updates are computed independently per PU without inter-unit communication. Upon completion, each PU returns its computed score array $f[]$ and the corresponding predecessor array $p[]$ to the host. In step 4, the host performs lightweight backtracking to reconstruct the optimal chain for each read. This post-processing step is embarrassingly parallel and introduces minimal overhead. Reads classified as anchor-sparse are processed entirely on the host using the original software chaining routine, avoiding unnecessary offloading and ensuring compatibility with existing alignment pipelines.

Furthermore, step 3 supports *dual-side execution*, enabling the host and PUs to operate concurrently. While anchor-sparse reads are processed on the host, anchor-dense reads are executed on PUs with higher parallelism and bandwidth. This

co-execution model prevents idle compute units and improves system-wide resource utilization under skewed workloads.

D. Implementation and Optimization

Implementation. We implement LAMP on a real NMP platform composed of host CPUs, standard DRAM DIMMs, and NMP-enabled DIMMs with integrated PUs [17]. Host-side logics are implemented in C, leveraging the UPMEM SDK [26] to control the NMP subsystem. This includes managing PU execution, orchestrating data transfers, and handling synchronization. Each PU is programmed in a low-level C-style language to control the RISC core and manage data-flow between the scratchpad and its local DRAM bank. To maximize intra-PU parallelism and pipeline utilization, all available hardware threads are activated for both indexing and chaining tasks [4]. We further optimize communication throughput by enabling parallel data transfers across all PUs within a rank. This achieves an aggregate bandwidth of up to 6.68 GB/s for host-PU transfers and 4.74 GB/s for PU-host transfers [4].

Workflow Optimization. To further improve execution efficiency, we implement a host-side multi-threaded pipeline that overlaps data preparation and computation across both CPU and PU domains. Specifically, one thread continuously reads query sequences from input files, while two dedicated threads handle task batching: one for host-side workloads and the other for PU-bound batches (step ③ in Figure 8). A shared queue is used to coordinate these threads. Once a batch of reads is parsed, each read is classified as anchor-dense or anchor-sparse. The batch-handling threads then independently execute their assigned workloads. Anchor-sparse reads are processed entirely on the host using the standard chaining implementation, while anchor-dense reads are dispatched to the PUs for parallel execution and synchronized result retrieval. This ensures that both host and PUs remain actively utilized, minimizing idle time and improving system throughput. Additionally, similar to Minimap2, LAMP supports a pipelined execution model where seeding, chaining, and alignment are overlapped across different reads. This pipelining strategy enhances throughput and can be integrated with LAMP’s scheduling model. A comparative evaluation of this design is presented in our experimental results section.

IV. EVALUATION

A. Experimental Setups

To show the feasibility of our approach, we evaluate LAMP on a real NMP platform as mentioned in Section III-D, the parameters are listed in Table I. All experiments are conducted on a system running Ubuntu 20.04.6 LTS. The host system is equipped with an Intel Xeon Silver 4110 CPU and 128 GB of DDR4 DRAM. The NMP subsystem contains 2048 PUs, each with dedicated instruction memory, scratchpad, and local DRAM for independent execution. For evaluation, we use the human reference genome Hg38 [18], which is approximately 3.2 Gb in size. The query dataset comprises approximately 185,000 reads, with minimum, maximum, and average read lengths of 44 bp, 95,479 bp, and 10,558 bp, respectively. This dataset provides a realistic workload for assessing the performance of both index construction and chaining stages,

especially under varying anchor distributions and workload imbalance. To quantify performance, we compare LAMP against the baseline CPU implementation of Minimap2 [1], focusing on execution time improvement.

TABLE I
SYSTEM CONFIGURATION USED FOR LAMP EVALUATION.

Host	
Processor	2 Intel Xeon Silver 4110 CPUs
Standard DIMMs	128 GB DDR4
NMP System [17]	
NMP DIMMs	16 modules, dual-rank, 16 chips/module, 8 PUs/chip
PU configuration	24 hardware threads; 350 MHz
PU local memory	64 KB scratchpad; 64 MB DRAM bank

We conduct three sets of experiments to evaluate LAMP’s performance tuning and design trade-offs. The first two focus on the indexing and chaining stages, respectively. For each stage, we fix the total memory capacity of the NMP subsystem and vary two parameters: (1) the portion of local DRAM allocated per PU, and (2) the number of active PUs. This allows us to analyze how memory granularity impacts runtime, data transfer overhead, and PU utilization. We also assess how combining the optimal configurations from both stages affects the overall end-to-end runtime. The third experiment set evaluates the proposed *Density-Aware Adaptive Dispatching* strategy during chaining. We vary the classification threshold that determines which reads are offloaded to PUs based on their anchor density. This sweep helps identify the configuration that best balances host and PU workloads for improved system-wide resource utilization.

B. Experimental Result

Indexing Performance. We first evaluate the effectiveness of the proposed Load-Spreading Hashing mechanism during the indexing stage. We fix the total index size and vary the number of PUs, which inversely adjusts the DRAM occupancy per PU from 64 MB down to 16 MB, as shown in Figure 9. We observe that the initialization time remains nearly constant regardless of the number of PUs. The PU-host transfer time decreases slightly, as each PU produces a smaller output and rank-level transfers proceed concurrently. Similarly, PU execution time benefits from increased parallelism and is reduced accordingly. However, host-PU transfer time increases as more PUs are activated. This is primarily due to the growing overhead of partitioning the input data, formatting buffers, and performing the load-spreading hashing procedure. These preprocessing costs become increasingly significant as the number of active ranks rises.

Takeaway II. Using approximately 64 MB of DRAM per PU achieves the best overall performance in the indexing stage. Although increasing the number of PUs reduces execution time per PU, the additional overhead introduced on the host side outweighs these gains. Consequently, we favor fewer PUs with larger local memory to balance efficiency and overhead.

Chaining Performance. We next examine the impact of PU memory allocation on chaining performance. As shown in Figure 10, we vary the per-PU DRAM allocation from 4 MB to 64 MB. Smaller allocations (e.g., 4–8 MB) incur substantial packing overhead due to the need to prepare a larger

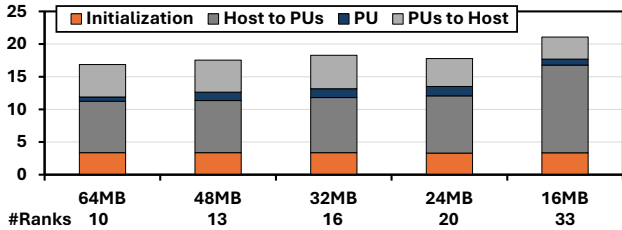


Fig. 9. Runtime breakdown of index construction under different PU memory occupancies (in seconds).

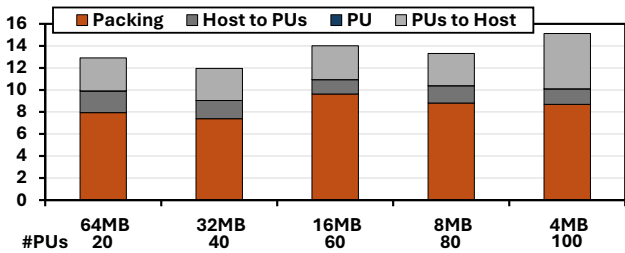


Fig. 10. Runtime breakdown of chaining under different DRAM occupancies per PU (in seconds).

number of data segments for more PUs. When using smaller batches, host-PU transfer time decreases due to improved rank-level parallelism. However, PU-host transfer time increases because the host must process a larger number of returned outputs. PU execution time improves modestly with more PUs, though its impact on total runtime is less pronounced due to its relatively small contribution to the total breakdown. Notably, chaining achieves high parallel efficiency as each PU executes independently without contention for shared memory.

Takeaway C1. The optimal configuration for chaining is 32 MB of DRAM per PU, with 40 active PUs. This setup balances data packing overhead, transfer efficiency, and parallel compute capacity. Very small DRAM allocations increase communication and parsing costs, while excessively large allocations reduce overall efficiency due to workload imbalance.

We also analyze the effectiveness of the density-aware adaptive dispatching strategy by varying the threshold that determines which reads are offloaded to PUs. As illustrated in Figure 11, low thresholds (e.g., “All PU” and 3,000) cause nearly all reads to be dispatched to PUs, which introduces rank-level execution imbalance. Since PUs operate at the rank level, all PUs within a rank must wait for the slowest unit to complete, causing performance degradation. High thresholds (e.g., 20,000 and “All CPU”) result in underutilization of PUs and miss the opportunity to accelerate memory-bound workloads. The best runtime is achieved at a threshold of 15,000 anchors per read, yielding a total chaining time of 66.1 s, representing a $3.6\times$ speedup over CPU-only execution and a $10.1\times$ improvement over PU-only execution.

Takeaway C2. Proper threshold tuning is essential for effective host-PU workload division. For our dataset, the 15,000 anchors/read threshold achieves the best trade-off, maintaining high utilization across both compute domains and minimizing idle time caused by anchor distribution skew.

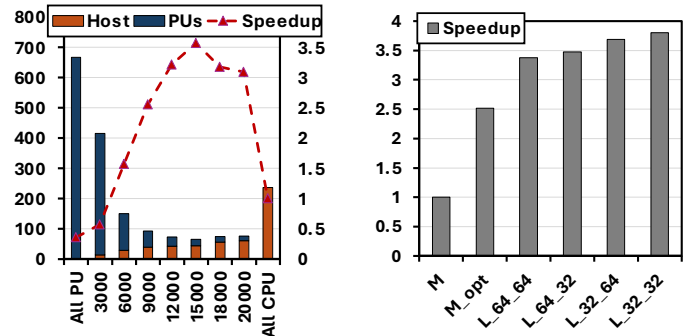


Fig. 11. Varying threshold of the density-aware adaptive dispatching (in seconds).

Fig. 12. Normalized runtime breakdown of index construction under different PU memory occupancies.

Overall System Performance. Figure 12 compares the end-to-end speedup of LAMP against the baseline Minimap2 (M) and the optimized multithreaded Minimap2 (M_opt) that overlaps seeding, chaining, and alignment in a pipeline. The L_X_Y labels indicate LAMP configurations, where X MB per PU is used for indexing and Y MB per PU is used for chaining. While M_opt achieves moderate speedup over the baseline by leveraging CPU-level threading, both baselines remain bottlenecked by memory-bound indexing and chaining. In contrast, LAMP consistently outperforms both baselines across all configurations. The highest speedup is around $3.8\times$ over M, which uses 32 MB per PU for both indexing and chaining. Other configurations with similar memory balances (e.g., L_64_64) also deliver strong performance, underscoring the importance of stage-specific tuning.

Takeaway O1. Optimal performance is achieved by selecting the best configuration for each pipeline stage. For our experimental setup, L_32_32 provides the best trade-off between PU count, local memory size, and system overhead, enabling LAMP to deliver near-maximum throughput.

V. CONCLUSION

This work presents LAMP, an adaptive near-memory processing framework designed to address the memory-bound and load-imbalance challenges in long-read mapping. By introducing a load-spreading hashing mechanism for balanced index construction and a density-aware adaptive dispatching strategy for efficient chaining, LAMP demonstrates that system-level co-design can integrate NMP hardware into existing genomic workflows. The design highlights the potential of near-memory architectures to improve scalability and throughput in data-intensive bioinformatics.

VI. ACKNOWLEDGMENTS

This work was supported in part by the National Science and Technology Council under grant nos. 112-2221-E-006-126-MY3, 114-2927-I-002-525, 114-2927-I-002-532, 114-2223-E-002-011, 114-2221-E-002-219-MY3, and 114-2221-E-002-222-MY3, and by the Ministry of Education under grant no. 114L900903.

REFERENCES

- [1] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.
- [2] F. J. Sedlazeck, P. Rescheneder, M. Smolka, H. Fang, M. Nattestad, A. Von Haeseler, and M. C. Schatz, "Accurate detection of complex structural variations using single-molecule sequencing," *Nature methods*, vol. 15, no. 6, pp. 461–468, 2018.
- [3] C. Jain, A. Rhie, N. F. Hansen, S. Koren, and A. M. Phillippy, "Long-read mapping to repetitive reference sequences using winnowmap2," *Nature methods*, vol. 19, no. 6, pp. 705–710, 2022.
- [4] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware," in *2021 12th International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2021, pp. 1–7.
- [5] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.
- [6] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.
- [7] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon *et al.*, "Near-memory processing in action: Accelerating personalized recommendation with axdim," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2021.
- [8] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An *et al.*, "System architecture and software stack for gddr6-aim," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE, 2022, pp. 1–25.
- [9] S. Chen, H. Tan, A. C. Zhou, Y. Li, and P. Balaji, "Updlrm: Accelerating personalized recommendation using real-world pim architecture," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.
- [10] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding future pim architectures by demystifying a commercial pim technology," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 263–279.
- [11] D. Lavenier, R. Cimadomo, and R. Jodin, "Variant calling parallelization on processor-in-memory architecture," in *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2020, pp. 204–207.
- [12] D. Lavenier, J.-F. Roy, and D. Furodet, "Dna mapping using processor-in-memory architecture," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2016, pp. 1429–1435.
- [13] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi *et al.*, "A case study of {Processing-in-Memory} in {off-the-Shelf} systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 117–130.
- [14] L.-C. Chen, C.-C. Ho, and Y.-H. Chang, "Uppipe: A novel pipeline management on in-memory processors for rna-seq quantification," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.
- [15] C.-L. Yeh, L.-C. Chen, C.-C. Ho, Y.-M. Chang, and D.-W. Chang, "Pimdud: An optimized deduplication design on a real processing-in-memory system," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–7.
- [16] L.-C. Chen, C.-C. Ho, and Y.-H. Chang, "Accelerating rna-seq quantification on a real processing-in-memory system," *IEEE Transactions on Computers*, 2025.
- [17] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, 2019, pp. 1–24.
- [18] Genome Reference Consortium. (2013) Grch38/hg38 - human genome reference. [Online]. Available: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.26
- [19] NCBI Sequence Read Archive. (2019) Sra dataset srr10326407. [Online]. Available: <https://www.ncbi.nlm.nih.gov/sra/SRR10326407>
- [20] D. Lee, B. Hyun, T. Kim, and M. Rhu, "Analysis of data transfer bottlenecks in commercial pim systems: A study with upmem-pim," *IEEE Computer Architecture Letters*, 2024.
- [21] I. Lee, B.-K. Wang, L.-C. Chen, W. S. Lim, D.-W. Chang, Y.-M. Chang, C.-C. Ho *et al.*, "Pim or cxl-pim? understanding architectural trade-offs through large-scale benchmarking," *arXiv preprint arXiv:2511.14400*, 2025.
- [22] Intel, "Intel Advisor," <https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html>, accessed: 2024-2025.
- [23] Intel, "Intel Developer Cloud," <https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html>, accessed: 2024-2025.
- [24] L.-C. Chen, S.-Q. Yu, C.-C. Ho, Y.-H. Chang, D.-W. Chang, W.-C. Wang, and Y.-M. Chang, "Rna-seq quantification on processing in memory architecture: Observation and characterization," in *2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2022, pp. 26–32.
- [25] S. Dahlgaard, M. B. T. Knudsen, E. Rotenberg, and M. Thorup, "The power of two choices with simple tabulation," in *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 2016, pp. 1631–1642.
- [26] "UPMEM SDK," <https://sdk.upmem.com/>, accessed: 2025.