Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching

Lingxi Wu
University of Virginia
lw2ef@virginia.edu

Rasool Sharifi
University of Virginia
as3mx@virginia.edu

Marzieh Lenjani
University of Virginia
ml2au@virginia.edu

Kevin Skadron
University of Virginia
skadron@virginia.edu

Ashish Venkat
University of Virginia
venkat@virginia.edu

***Abstract*—The rapid influx of biosequence data, coupled with the stagnation of the processing power of modern computing systems, highlights the critical need for exploring high-performance accelerators that can meet the ever-increasing throughput demands of modern bioinformatics applications. This work argues that processing in memory (PIM) is an effective solution to enhance the performance of k-mer matching, a critical bottleneck stage in standard bioinformatics pipelines, that is characterized by random access patterns and low computational intensity.**

This work proposes three DRAM-based in-situ k-mer matching accelerator designs (one optimized for area, one optimized for throughput, and one that strikes a balance between hardware cost and performance), dubbed Sieve, that leverage a novel data mapping scheme to allow for simultaneous comparisons of millions of DNA base pairs, lightweight matching circuitry for fast pattern matching, and an early termination mechanism that prunes unnecessary DRAM row activation to reduce latency and save energy. Evaluation of Sieve using state-of-the-art workloads with real-world datasets shows that the most aggressive design provides an average of 326x/32x speedup and 74X/48x energy savings over multi-core-CPU/GPU baselines for k-mer matching.

***Index Terms*—Processing-in-memory, Bioinformatics, Accelerator**

I. Introduction

The field of bioinformatics has enabled significant advances in human health through its contributions to precision medicine, disease surveillance, population genetics, and many other critical applications. The centerpiece of a bioinformatics pipeline is genome sequence comparison and classification, which involves aligning query sequences against reference sequences, with the goal of identifying patterns of structural similarity and divergence. While traditional sequence alignment algorithms employ computationally-intensive dynamic programming techniques, there has been a growing shift to a high-performance heuristic-based approach called *k-mer matching*, that breaks a given query sequence into a set of short subsequences of size k, which are then scanned against a reference database for hits, with the underlying assumption that biologically correlated sequences share many short lengths of exact matches. K-mer matching has been deployed in a wide array of bioinformatics tasks, including but not limited to, population genetics [1], cancer diagnosis [2], metagenomics [3]–[8], bacterial typing [9], and protein classification [10]. K-mer matching may also show up in other application domains, but in this paper, we focus on bioinformatics.

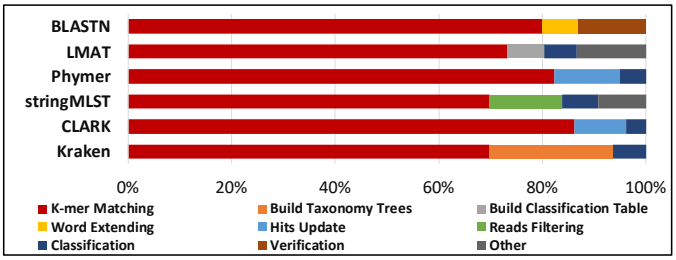

Fig. 1: **Execution time breakdown of Kraken [3], CLARK [6], stringMLST [9], Phymer [1], LMAT [7], BLASTN [15]**

The acceleration of bulk k-mer matching is of paramount importance for two major reasons. First, k-mer matching sits on the critical path of many genome analysis pipelines. Figure 1 shows the execution breakdown of several important bioinformatics applications that target a variety of tasks ranging from metagenomics to population genetics, and clearly, k-mer matching dominates the execution time in all applications. Second, modern sequencing technologies have been shown to generate data at a rate surpassing Moore's Law [11]. In fact, by 2025, the market share of metagenomics alone is expected to reach $1.4 billion, and the amount of data that needs to be analyzed by metagenomics pipelines is projected to surpass that of YouTube and Twitter [12]. To further exemplify the scale of data explosion and processing overhead, consider the case of precision medicine, where a patient's sample can be sequenced in roughly 48 hours on the NovaSeq instrument, producing 10 TB of microbiome and DNA/RNA data [13]. To develop personalized treatment from these samples, raw sequences are passed through, often in parallel, various metagenomics stages with k-mer matching on the critical path (e.g., ∼68 days on Kraken [3]). These tasks play a critical role in combating pandemics and treating antibiotic-resistant infections, saving billions of dollars in health care costs [13], [14].

However, despite its significance, the acceleration of k-mer matching on modern high-end computing platforms remains a challenge, due to its inherently memory-bound nature, considerably limiting downstream genome analysis tasks from realizing their full potential. In particular, k-mer matching algorithms are typically characterized by random accesses across large memory regions, leading to poor cache behavior, even on high-end servers that feature large last-level caches. The cache-unfriendliness of k-mer matching will continue to get worse with the rapid growth in the size and complexity

of genomic databases, making the task a major bottleneck in modern bioinformatics pipelines. This is further exacerbated by the fact that the computation per k-mer lookup is too small to mask the high data access latency, thereby rendering existing compute-centric platforms such as multi-core CPUs and GPUs inadequate for large-scale genome analysis tasks.

Memory-centric solutions to accelerate bioinformatics applications come in a variety of flavors, but recent proposals demonstrate that *near-data* [16]–[18] and *in-memory processing* systems [19]–[21] have promising potential to improve the efficiency of large-scale genome analysis tasks, owing to the fact that these applications are increasingly characterized by their high data movement (from memory to the processor) and low computation (within the processor) costs [22].

This work explores the design space for high-performance k-mer matching accelerators that use logic in DRAM as the basis for acceleration, including the most aggressive form of *processing-in-memory* (PIM), in-situ computing, with the goal of parallel processing of sequence data within DRAM row buffers. To this end, we propose Sieve, a set of novel **S**calable **i**n-situ DRAM-bas**e**d accelerator designs for massi**v**ely parallel k-m**e**r matching. Specifically, we offer three separate designs: Sieve Type-1, Type-2, and Type-3. Each architecture incrementally adds extra hardware complexity to unlock more performance benefits. Note that, although our approach involves modifying conventional DRAM organization, we do not propose change conventional DRAM; our goal is to only leverage DRAM technology to build a new accelerator. Ultimately, the value of the accelerator will determine whether a new DRAM-based chip is worth the design and manufacturing effort.

The advantage of in-situ computing is that the bandwidth at the row buffer is six orders of magnitude larger than that at the CPU, while the energy for data access is three orders of magnitude lower [23], [24]. However, in-situ computing also introduces several key challenges. First, in-situ acceleration necessarily requires the tight integration of processing logic with core DRAM components, which has been shown to result in prohibitively high area overheads [19], [21]. In fact, even a highly area-efficient state-of-the-art in-situ accelerator is only half as dense as regular DRAM [19]. However, bioinformatics applications typically favor accelerators with larger memory capacity due to their ability to accommodate the ever-increasing DNA datasets that need to be analyzed within short time budgets. Second, existing in-situ approaches [19], [20] rely on multi-row activation and row-wise data mapping to perform bulk Boolean operations of data within row buffers, resulting in substantial loss of throughput and efficiency [21]. Finally, to capitalize on the performance benefit of in-situ computing for k-mer matching, it is imperative that the accelerator is provisioned with an efficient k-mer indexing scheme that avoids query broadcasting, and a mechanism to quickly locate and transfer payloads (e.g., genome taxon records).

Key Contributions. The distinguishing feature of Sieve is the placement of reference k-mers vertically along the bitlines of DRAM chips and subsequently utilizing sequential single-row activation rather than the multi-row activation proposed in prior works, to look up queries against thousands of reference k-mers simultaneously. The column-wise placement of k-mers further allows us to employ a novel Early Termination Mechanism (ETM) that interrupts further row activation upon the successful detection of a k-mer mismatch, thereby considerably alleviating the latency and energy overheads due to serial row activation. To the best of our knowledge, this is the first work to introduce and showcase the effectiveness of such a column-wise data mapping scheme for k-mer matching with early termination, substantially advancing the state-of-the-art in terms of both throughput and efficiency.

By exploiting the fact that matching individual k-mers is relatively less complex than most other conventional PIM tasks such as graph processing, in this work, we design a specialized circuit for k-mer matching, with the goal of minimizing the associated hardware cost. We then meticulously explore the design space of in-situ PIM-based accelerators by placing such custom logic at different levels of the DRAM hierarchy from the chip I/O interface (Type-1) to the subarray level (Type-2/3), with a detailed analysis of the performance-area-complexity trade-offs, and a discussion of system integration issues, deployment models, and thermal concerns. We compare each Sieve design with state-of-the-art k-mer-matching implementations on CPU, GPU, and FPGA, and perform rigorous sensitivity analyses to demonstrate their effectiveness. We show that the processing power of Sieve scales *linearly* with respect to its storage capacity, considerably enhancing the performance of modern genome analysis pipelines. Our most aggressive design provides an average speedup of 210X/35X and an average energy savings of 35X/71X over conventional multi-core-CPU/GPU baselines

II. Background

In this section, we introduce the k-mer matching procedure and explain why it is a bottleneck stage in conventional architectures.

K-mer Matching in Bioinformatics. A DNA sequence is a series of nucleotide bases commonly denoted by four letters (bases): A, C, G, and T. K-mers are subsequences of size k. Metagenomic algorithms attempt to assign taxonomic labels to genetic fragments (sequences) with unknown origins. A "taxonomic label" is an assignment of a sequence to a particular organism or species. Traditionally, this is done by aligning an individual query sequence against reference sequences, which can be prohibitively slow. Processing a metagenomics file containing 10^7 sequences using an alignment-based BLAST algorithm takes weeks of CPU time [25], [26]. Experts predict that genomics will soon become the most prominent data producer in the next decade [11], demanding more scalable sequence analysis infrastructure. More recently, alignment-free tools that rely on simple k-mer matching have emerged to aid large-scale genome analysis tasks, due to the fact that properly labeled k-mers are often sufficient to infer taxonomic and functional information of a sequence [3], [6], [7], [27].

Figure 2 illustrates the process of a typical k-mer-matching-based sequence classifier. In an offline stage, a reference k-

```
1.   for (query_seq: query_list){
2.       kmer_list = []
3.       payload_list = []
4.       ... // store k-mers from query_seq
5.       for (kmer: kmer_list){
6.          result = query_kmer(kmer, reference k-mer set, ...)
7.          if (result != NULL) // found match, retrieve payload
8.             payload_list.add(result.payload)
9.          else
10.            ... // no match
11.      }
12.      ... // classify query_seq using payload_list
13.  }
```

Fig. 2: **k-mer matching-based sequence classification.**

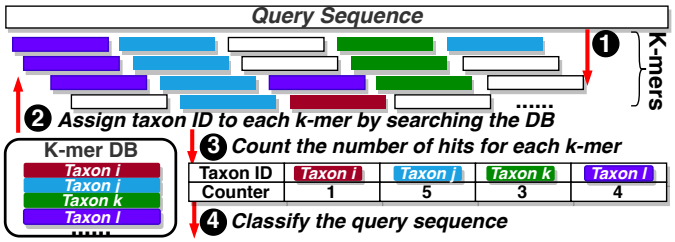

Fig. 3: **Illustration of CLARK [6], a state-of-the-art k-mer matching based metagenomics tool. A taxon label is a formal scientific name to identify bacterial, fungi, virus, and other species.**

mer database is built, which maps unique k-mer patterns to their taxon labels. For example, if a 5-mer "AACTG" can only be found in the E.coli bacteria sequence, an entry that maps "AACTG" to E.coli is stored. At run time, k-mer matching algorithms slide a window of size k across the query sequence, and for each resulting k-mer, they attempt to retrieve the associated taxon label from the database. Function **query_kmer** in line 6 is repeatedly called to search each k-mer in the database. If the query k-mer exists in the database (k-mer hit), its taxon label (payload) is retrieved, otherwise we move on and compare the next k-mer in the query. Once all k-mers in a query are processed, the taxon labels of the matched k-mers are used to make a final decision on the originating organism for the query sequence. A popular choice is to keep a counter for each retrieved taxon label, and the taxon label with the most hits is used to classify the query sequence. See Figure 3 for example. The reference k-mer set itself can be implemented in a number of ways. CLARK [6] and LMAT [7] leverage a hash table, with the k-mer pattern as the key and the taxon label as the value. Kraken [3] uses a more sophisticated data structure that is a hybrid between a hash table and a sorted list, in which k-mers that share the same "signature" are put into the same hash bucket, which is then looked up using binary search. The assumption here is that two adjacent k-mers within a query sequence are likely to share the same "signature", since they overlap by (k-1) bases, and are thereby likely to get indexed into the same bucket. In theory, this improves the cache locality over purely hash table or sorted list approaches, since matching the first query k-mer often brings the bucket to the cache which will be used for the subsequent query k-mers. As we note below, even with this optimization, cache performance remains poor.

Memory Is the Bottleneck for K-mer Matching. Real-world k-mer matching applications expose limited cache locality. For sequence classifiers that store reference k-mers in a hash table, accessing a hash table generates a large number of cache misses due to the linked list traversal or repeated hashes (to resolve hash collision). While a hash table/sorted list hybrid can provide better locality, since the k-mer bucket can be fetched into the cache from the previous k-mer lookup, using Kraken and its supplied datasets, we discover that only 8% of consecutive k-mers index into the same bucket, resulting in new buckets fetched repeatedly from memory to serve requests. k-mer matching also benefits from finer-grained memory access—k-mer records are typically around 12 bytes [3], while each memory access retrieves a cache line of data, which usually serves only one request due to poor locality, resulting in waste of bandwidth and energy. Finally, computational intensity of k-mer matching is too little to mask extended data access latency. Using CLARK (Figure 3) as an example, we find that while retrieving k-mers from a database takes many cycles due to cache misses, updating counters for matched k-mers is trivially inexpensive, amplifying the effects of the memory wall [22].

III. Motivation and Key Ideas

In this section, we address the main challenge of designing in-situ k-mer matching accelerators, namely integrating logic into DRAM dies with low hardware overhead. We propose three separate Sieve designs to combat this issue. We then identify the key limitations of prior in-situ work when adapted for k-mer matching and motivate our novel data layout and pattern matching mechanisms. Finally, we introduce an Early Termination Mechanism (ETM) to further optimize Sieve by exploiting characteristics of real-world sequence datasets.

DRAM Overhead Concerns. While in-situ accelerators can provide dramatic performance gains for memory-intensive applications, building them with reasonable area overhead is difficult [19], [21]. The sense amplifiers in row buffers are laid out in a pitch-matched manner, and the DRAM layout is carefully optimized to provide high storage density, and fitting additional logic into the row buffer in a minimally invasive way is non-trivial. Moreover, since the number of metal layers of a DRAM process is substantially smaller than that of the logic process, building complex logic with a DRAM process incurs significant interconnect overhead [19], [21].

We design a set of core k-mer matching operations for Sieve using simple Boolean logic. Sieve has very little hardware overhead compared to other PIM architectures, because k-mer matching, which is mainly accomplished by exact pattern matching, can be supported by a minimal set of Boolean logic.

Trade-offs of Different Sieve Designs. To explore optimal Sieve designs, we compare the placement of custom k-mer matching logic at three different levels in the DRAM hierarchy: from the I/O interface of the DRAM chips (Sieve Type-1) to the local row buffer of each subarray (Sieve Type-

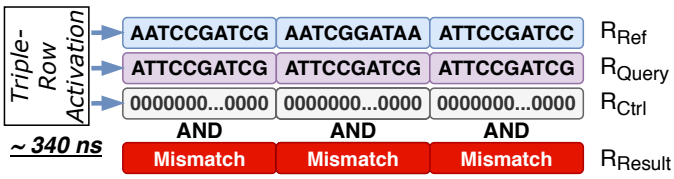

Fig. 4: **K-mer matching in existing in-situ accelerators using Triple-row Activation and horizontal data layout.**

Fig. 5: **K-mer matching in Sieve using Single-Row Activation and vertical data layout.**

3), and Type-2 as the middle ground where several subarrays share one k-mer matching unit. Recall that a DRAM bank's transistor layout is highly optimized for storage, and inserting extra logic, however minimal, requires significant redesign effort. Type-1, illustrated in Figure 12, keeps the bank layout intact, and thus is the least intrusive design. However, it suffers from the lowest parallelism and the highest latency, because the comparison is restricted to a column of bits rather than the entire row. Sieve Type-2 increases parallelism and energy efficiency over Type-1 by accessing a row of bits. Type-3 leverages subarray-level parallelism (SALP) [28], providing the highest performance potential, but it comes at the cost of the highest design complexity and hardware overhead.

Novel Data Layout and Pattern Matching Mechanism. We show that our column-wise k-mer data layout and row-wise matching mechanism, combined with *early termination* outperforms prior in-situ accelerators that rely on multi-row activation and conventional row-wise mapping. The majority of the k-mer matching workload is exact pattern matching, which can be performed using bulk bitwise XNOR between two operand DRAM rows. The prior arts such as Ambit and DRISA implement XNOR operation by first ANDing two rows along with a third control row (populated with 1s or 0s), and send the results to an additional logic. In the following analysis, we only consider the timing delay of the AND operation to give advantage to the previous in-situ PIM work. Ambit [20] is used as a baseline. Both Ambit and *1T1C-based* DRISA [19] are inspired by the same work [29] for in-situ AND procedure. Thus, their performance for k-mer pattern matching is similar. Ambit performs bulk bitwise AND in reserved DRAM rows (see Figure 4). Assuming a DNA base is encoded with two bits (by NCBI standard [30]), a common k value of 31, and a typical DRAM row width of 8192 bits, then each row fits 128 k-mer patterns if k-mers are stored in a row-wise manner. To search a query against a group of references, Ambit first copies 128 reference patterns from the data region to R_{Ref}. It then makes 128 copies of the same query in R_{Query}. Since the target operation is AND, the control row (R_{Ctrl}) is populated with 0s (copied from a preset row). Next, a triple-row activation is performed on R_{Ref}, R_{Query}, and R_{Ctrl}. Finally, the result bits are copied to another row R_{Result}. One row-wide AND takes 8 row activations and 4 precharge commands from setting up to completion, which is $8 \times tRAS + 4 \times tRP =\sim 340ns$ for a typical DRAM chip.

In contrast to these approaches, ComputeDRAM [31] enables in-memory computation in commodity DRAMs, without the need for integrating any additional circuitry. The key to this approach is the fact that issuing a constraint-violating sequence of DRAM commands in rapid succession leaves multiple rows open simultaneously, allowing row-wide copy, logical AND, and logical OR operations to be performed via bit line charge sharing, essentially free of hardware cost.

While all of these approaches can be leveraged to perform k-mer matching, our analysis suggests that significant gains in performance and energy efficiency can be achieved by employing the column-major approach we propose in this work, that not only eliminates the need for multi-row activation, but also enables a synergistic early termination mechanism that inhibits further row activations upon finding a match.

More specifically, Sieve does not compare a full-length query k-mer against a set of full-length reference k-mers at once. Instead, it compares a query with a more extensive set of references in a shorter time window ($1 \times tRAS + 1 \times tRP =\sim 50ns$), but progresses only one bit at a time (see Figure 5). Reference bits in Sieve are laid out column-wise, along bitlines. Thus, a single row activation transfers 8K bits into the matchers embedded in row buffers for comparison. Each matcher has a one-bit latch to keep track of the matching result. The next row is activated, and a new batch of reference bits is compared, until ETM (introduced next) interrupts when all latches return zero.

Processing only one bit at a time does not hurt Sieve's performance, because it leverages parallelism across the rows; i.e., it performs 8K comparisons at once. The vertical data layout greatly expands the initial search space (128 reference k-mers to 8192 reference k-mers), and our *early termination mechanism* (ETM) quickly eliminates most of the candidates after just a few row activations. Besides the latency reduction for each row-wide pattern matching by adopting single-row activation (~340 ns to ~50 ns), Sieve also reduces activation energy, since raising each additional wordline increases the activation energy by 22% [20]. Thus, even if the same data mapping strategy is applied, the multi-row activation-based approach is still slower and less energy efficient than Sieve simply because of the internal data movement. Note that the internal data movements associated with multi-row activation is unavoidable, because the operand rows have to be copied to the designated area. Furthermore, arbitrarily activating three rows inside the DRAM requires a prohibitively large decoder (possibly over 200% area overhead [19]), and activating more than one row could potentially destroy the original values.

The Motivation for Early Termination. Activating consecutive rows in the same bank results in highly unfavorable DRAM access patterns that are characterized by long delays (due to more row cycles) and high energy costs (row opening

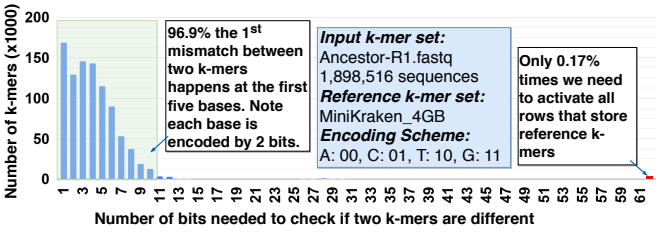

Fig. 6: **Characterization of mismatches between k-mers.**

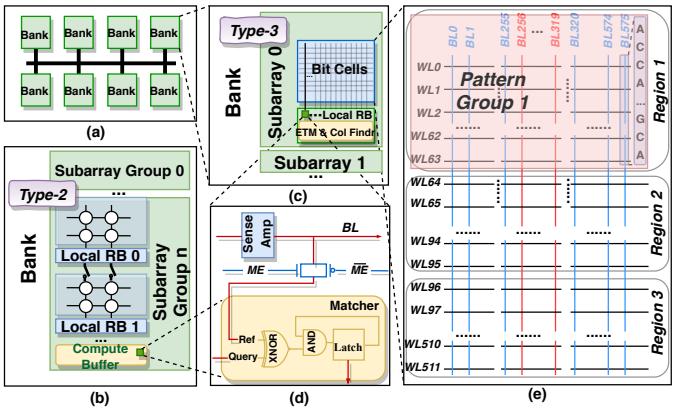

Fig. 7: **Sieve Overview. (a) DRAM banks. (b) Type-2 Zoom-in. Subarray group facilitates inter-subarray data copy, and a compute buffer is added for each subarray group which has the matcher circuits. (c) Type-3 Zoom-in. Similar to Type-2 but the matchers reside in the local row buffers. (d) Matcher. (e) Data layout of subarray. Each subarray is partitioned into three regions for storing k-mer pattern groups, payload offsets, and payloads.**

dominates DRAM energy consumption [32]).

We identify a novel optimization opportunity that exploits the concept of the *Expected Shared Prefix* (ESP), which describes the first mismatch location between two random sequences. On average, for DNA sequences between 1k and 16k bases, the first mismatch is known to occur between the sixth and the eighth base [33]. The ESP is even smaller than six for short k-mers, as shown in in Figure 6. For random k-mers extracted from metagenomics reads, when matched against reference k-mers, 97% of the first mismatch can be found within the first five bases (first 10 bits if each base is encoded by two bits).

IV. Sieve Architecture

This section describes the three Sieve designs. We introduce Types-2 and 3 first, as they exploit greater parallelism, and follow it up with Type-1 due to difference in design details.

A. Sieve Type-2 and Type-3

Figures 7 (b) and (c) show the functional block diagrams of Type-2/3. They differ mainly in the placement of the add-on logic (e.g., matching circuitry) at the bank vs. subarray level, but share the same data mapping scheme.

Data Layout. K-mer patterns are encoded in binary (A: 00, C: 01, G: 10, T: 11) and transposed onto bitlines, for

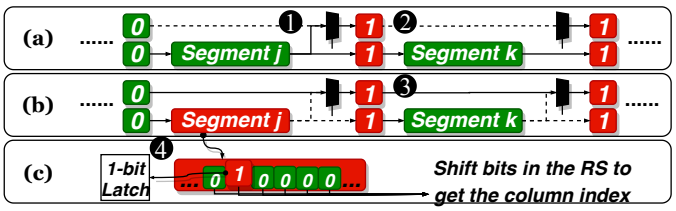

Fig. 8: **Column Finder in Type-2/3. Segments with k-mer hits are shown in red, otherwise green.**

column-wise placement, as described in the previous section. Bit cells within each subarray are divided into three regions (Figure 7 (e)). However, we note that no physical modification is made to the bit cells. Region-1 stores the interleaved reference and query k-mers. Region-2 stores the offsets to the starting address of payloads (one for each reference k-mer), allowing us to precisely locate the payloads. Region-3 stores the actual payloads such as taxon labels. Data in Region-2/3 is stored in conventional row-major format. The main motivation to co-locate patterns and payloads is to minimize contention and achieve higher levels of parallelism. If patterns are densely packed into several dedicated banks/subarrays, all matching requests will be routed to them, creating bank access contention and serializing such requests.

Region-1 is further broken down into smaller pattern groups and a batch of 64 (different) query k-mers are replicated in each pattern group in the middle (red in Figure 7(e)). This is because the transmission delay of long wires inside DRAM chips prevents us from broadcasting a query bit to all matchers (discussed next) during one DRAM row cycle. All pattern groups in a subarray work in the lockstep manner. The exact size of a pattern group is equivalent to the number of matchers that a query bit can reach in one DRAM row cycle. In this example (DDR3_micron_32M_8B_x4_sg125), it happens to be 576 (*512 reference k-mers + 64 query k-mers*). The number of query k-mers per batch is determined by the chip's prefetch size. In this example, a chip with a prefetch size of 8 bytes writes 64 bits with a single command. A chip with smaller (larger) prefetch size has smaller (larger) batch size. After a batch of query k-mers finishes matching in a subarray, they are replaced by a new batch. The total number of write commands needed to replace a batch of 64 k-mers can be computed as *(# of pattern groups / subarray) × (k × 2)*.

Matcher. We enhance each sense amplifier in a row buffer with a matcher shown in Figure 7 (d). The matcher of Type-2/3 is made of an XNOR gate, an AND gate, and a one-bit latch. The XNOR gate checks if the reference bit and the query bit at the current base are equal. The bit latch stores the result of the XNOR operation, indicating if a reference and a query have been matched exactly up until the current base. The value in each bit latch is set to 1 initially (default to match). The AND gate compares the previous matching result stored in the bit latch with the current result from the XNOR gate and updates the bit latch accordingly, capturing the running outcome bit-by-bit. Finally, we allow the matcher to be bypassed or engaged by toggling the Match Enable signal.

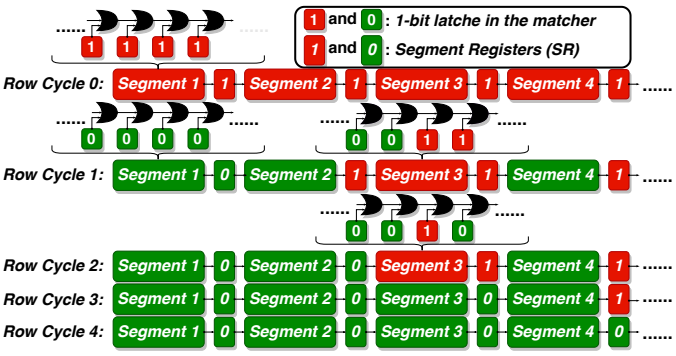

Fig. 9: **ETM in Type-2/3.**

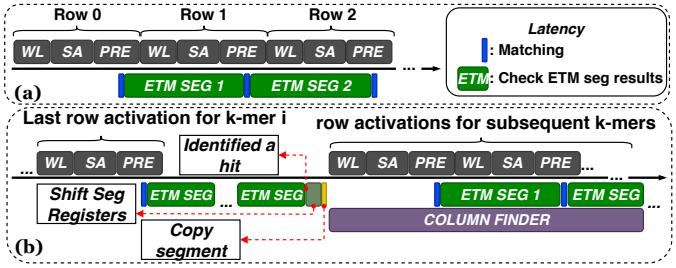

Fig. 10: **Type-3 Timing Analysis. WL, SA, and PRE indicate latencies associated with raising the wordlines, enabling sense amplifiers and precharging the rows. (a) ETM and matchers operations overlap with row opening. (b) ETM is on the critical path only when there is a hit, as it needs extra cycles to identify the hit. Then the BSRs are shifted, followed by a copy into the RS. CF operates in parallel with row opening and ETM for the next k-mer.**

When a row is opened, both query and reference bits are sent to sense amplifiers. A subarray controller [19] (sCtrl) then selects which query to process among the 64 queries in the subarray. Each pattern group has a 1-bit shared bus connecting all matchers. The selected query bit is distributed to all matchers in a pattern group through this shared bus.

Early Termination Module (ETM). The ETM interrupts further row activation by checking if the entire row of latches is storing zeros. The k-mer matching process continues if at least one latch stores 1. The natural way is to OR the whole row of latches. However, the challenge of this approach is that each OR gate adds to the latency, and during one DRAM row cycle, only a small fraction of result latches can propagate their results through OR gates. We propose a solution that breaks the row of latches into segments and propagates partial results in a *pipelined* fashion. (shown in Figure 9). One segment register (SR) is inserted for every 256 latches to implement the pipeline. During one DRAM row cycle, each segment takes the value from the previous SR, ORs it with all its latches, and outputs the value to the next SR. Notice that in Figure 9, although at row cycle 3, all latches store zeros, the last SR still holds 1. An extra cycle is needed to flush the result

Column Finder (CF). Unless interrupted by the ETM, the row activation continues until all bases of a query are checked. If a query is previously matched to a reference, one and only one latch in a row buffer stores one. The Column Finder identifies the column (bitline) that is connected to that latch. The column numbers are needed to retrieve offsets, and subsequently, payloads. Our solution is to shift a row of latched bits until we find a one. The challenge of this approach is to design a shifter with reasonable hardware cost and latency. In the worst case, where the matched column (reference k-mer) is located at the end of the row, the CF needs to shift an entire row of latched bits. We propose a pipelined, two-level shifter solution for CF. Figure 8 illustrates this. The CF circuits are re-purposed mainly from those of the ETM. For each ETM segment, a MUX ❶ and a 1-bit Backup Segment Register (BSR) ❷ are added (Figure 8 (a)). BSRs and SRs maintain the same values and are updated simultaneously during the ETM operation. Zero in a BSR means that its associated segment does not contain a match, and one implies it does. Further, we add another set of bit latches called the Reserved Segment (RS) shown in Figure 8 (c), which includes the same amount of 1-bit latches and OR gates as a segment.

For Column Finder, the BSRs are first shifted until we find a one, to narrow down the appropriate segment that contains a match (❸ in Figure 8 (b)). We then copy this segment over to the Reserved Segment (RS) where the final round of shifting happens ❹. From this point on, all ETM segments are freed to support the pattern matching for the next k-mer, while the CF works in the background to retrieve the column number (see Figure 10 (b)). The shifting of bits in RS is overlapped with the matching of the subsequent k-mer. We point out two details here. First, after the last row activation for a given query k-mer finishes, ETM takes up to 256 DRAM row cycles to flush the pipeline in the worst case, when the one is at the very end. During this time, no new row activation is issued, and the CF operation is stalled until ETM completes. Second, note that each k-mer hit takes up to 4800 DRAM cycles, while the CF operation takes up to 1032 DRAM cycles in the worst-case scenario. Therefore, we observe no contention at the CF, even when there are two consecutive hits in the same subarray.

Sieve Type-2. While Type-2 retains most of the high-level design from Type-3 (ETM, data mapping, matching circuits, etc.), it differs in one key aspect – instead of integrating logic to all subarrays at the local row buffer level, logic is added to a *subarray group* – a subset of adjacent subarrays within a bank (e.g., 1/2, 1/4, 1/8 of subarrays) connected through high bandwidth links (isolation transistors). Each subarray group is equipped with a *compute buffer*, which retains much of the capabilities (k-mer matching, ETM, and column finding) of a local row buffer in Type-3 without its sense amplifiers. Unlike type-3, where k-mer matching is performed locally at each individual subarray, Type-2 processes k-mer matching inside the compute buffer regardless of the target subarray query k-mers get dispatched to. This involves transferring a row of bits across subarrays to reach the compute buffer at the bottom of the subarray group. To enable fast row copy across subarrays, we leverage LISA [34], albeit adapted to the folded-bitline architecture that Sieve is built upon. We validate the feasibility of our design with a detailed circuit-level SPICE simulation.

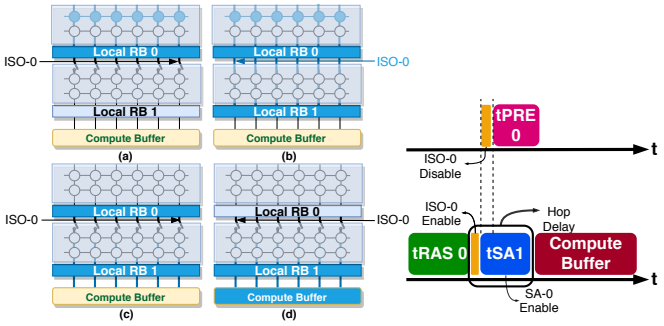

Fig. 11: **Row-wide data copy across subarrays.**

Figure 11 illustrates the process of transferring a row from the source subarray to its compute buffer – (a) the DRAM row in the subarray 0 is activated and the data is latched onto its local sense amplifiers, (b) when the bitlines of subarray 0 are fully driven, the links between the subarray 0 and subarray 1 are enabled. Due to charge sharing between the bitlines of subarrays 0 and 1, the local sense amplifiers in the subarray 1 senses the voltage difference between the bitlines and amplifies it further, as a result of which, (c) local sense amplifiers in both subarrays 0 and 1 start driving their bitlines to the same voltage levels, and finally, (d) when both sets of bitlines in subarrays 0 and 1 reach their fully driven states, the isolation transistors between them are disconnected and the local sense amplifiers in the subarray 0 are precharged. The process is repeated until the data reaches the computed buffer. Note that – (1) only two sets of local sense amplifiers are enabled at any time in a bank, and (2) as validated in our Spice simulation, the latency of activating the subsequent sense amplifiers (tSA in Figure 11 is much smaller ($\sim$ 8X) than activating the ones of the source subarray (tRAS). The latency for one row to cross a subarray (except for the first one) is referred to as "hop delay" which consists of enabling the isolation transistors (link) and the activation of the sense amplifiers.

K-mer Matching Walkthrough. We use Type-3 as an example to illustrate the k-mer matching process. Once a row is selected for activation, both the query and the reference bits are sent to the local row buffer for comparison using the mechanisms described above. The ETM checks all segments and propagates the values of Segment Registers (SRs) to determine if a match is found.Once a match is found, the payload associated with that k-mer pattern is retrieved as follows. The CF first determines the segment number by shifting all BSRs. It then gets the column index by shifting all 1-bit latches in that segment until the one is found. The column number is calculated as *segment_number* $\times$ *(# of columns / segment)* + *column_index* and sent to subarray controller to index into the payload address offsets.

B. Sieve Type-1

Sieve Type-1 is not a quintessential in-situ architecture, due to the lack of processing unit embedded in row buffers. However, Type-1 preserves the overall high-level ideas, such as the data layout, ETM, and the matching unit. In addition, Type-1 is the least intrusive implementation of Sieve because

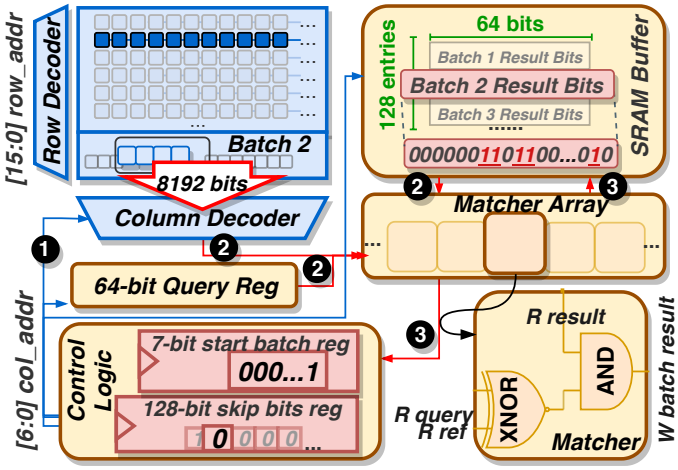

Fig. 12: **Sieve Type-1. A query k-mer is sent to the Query Register, and a row activation is issued. 1. The controller logic uses the column address to select a batch and indexes into the SRAM Buffer to get the batch result bits entry. 2: The query bit, the reference bits, and the result bits are sent to the Matcher Array. 3: Matchers write back to the result bits entry stored in the SRAM Buffer.**

it does not change the physical layout of DRAM banks. The bank I/O width is 64 bits, and each row is 8192 bits. Thus, a row is divided into 128 *batches*. A batch is a set of bits retrieved by a DRAM read burst of a read command. Batch size varies depending on the column width, which can be 32, 64, or 128 bits. Next, we introduce each component of Type-1.

SRAM Buffer (SB). SB stores the match result bits, organized in a 2D array. The number of entries is equal to the number of batches, and the entry width is the batch size. Before matching, all batch result bits are preset to one, and are updated as the matching progresses, again capturing the running match outcome. Figure 12 highlights the result of batch two, where zero indicates a mismatch.

Matcher Array (MA). MA consists of 64 matching units. It compares a query bit with the reference bit using an XNOR gate, and updates (writes back) the result bit by ANDing the match result bit stored in SB with the output from XNOR.

Skip Bits Register (SkBR). SkBR is used for ETM. It contains one bit for each batch indicating if we need to process the current batch. All bits in SkBR are preset to one. As the matching progresses, more and more bits in SkBR is set to zero, meaning more and more batches will be skipped. Without SkBR, each row activation is followed by 128 batch comparisons. Since most comparisons result in mismatches, SkBR leads to significant energy and latency reduction.

Start Batch Register (StBR). StBR reduces processing time further. Due to the ETM, Type-1 checks the skip bits to find proper batches to send to the MA. The search time is one DRAM cycle per skip bit. In the worst case where only the last batch is valid, 127 DRAM cycles are wasted to check all the previous skip bits. With the help of the StBR, whose value points to the first batch that needs to be processed, Type-1 can quickly determine the first batch to open.

Column Finder and Payload Retrieval. The control logic first checks the skip bits to locate the batches that contain a one, given the one-to-one mapping between batches and skip bits. A small shifter is applied to get the index of the matched column in the batch. The column number is calculated as (batch index) * (batch size) + (column index), and is then used by the control logic to get offsets and payload.

C. System Integration

We consider both Dual-Inline Memory Module (DIMM), and PCIe form factors for integrating Sieve into a host. While PCIe incurs extra communication overhead due to packet generation, DIMM suffers from limited power supply. A typical DDR4 DIMM provides around 0.37 Watt/GB [35] of power delivery and 25 GB/s of bandwidth, which is sufficient for Type-1. To satisfy the bandwidth and power requirement, Type-2 needs at least PCIe 3.0 with 8 lanes, and Type-3 needs at least PCIe 4.0 with 16 lanes.

We use a 32 GB Type-2 Sieve to illustrate how Sieve communicates with the host using a PCIe interconnect. Unlike Type-1, which communicates with the host on individual k-mer requests, Type-2/3 uses a packet-based protocol that delivers hundreds of k-mer requests per PCIe packet. A PCIe Type-2/3 accelerator maintains a (*PCIe Input Queue*) and a (*PCIe Out Queue*) for sending/receiving PCIe packets, and a *response ready queue (RRQ)* to hold serviced k-mer requests. The CPU scans the query sequences to generate k-mers, and for each k-mer, it makes a 12-byte *request* that contains the pattern, sequence ID, destination subarray ID, and other header information. Each PCIe packet contains 340 requests, assuming 4 KB PCIe packet payload size. Each Sieve bank buffers 64 requests. To fully saturate the capacity of a 32 GB Sieve, the depth of the PCIe queue is set to 24 (24 *PCIe packets* $\times$ 340 *requests / packet* $\approx$ 16 *ranks* $\times$ *8 banks / rank* $\times$ 64 *requests / bank*). Sieve removes the PCIe packets from *PCIe Input Queue*, unpacks them, and distributes requests to the target banks. A finished request gets moved to the *RRQ*. Once the *RRQ* is full, a batch of PCIe packets is moved to the *PCIe Out Queue*. Sieve sends an interrupt to the CPU if the packets are waiting in the *PCIe Out Queue* or if there are empty slots in the *PCIe Input Queue*.

The entire space of Sieve is memory-mapped to host as a noncacheable memory region, avoiding virtual memory translation and cache coherence management. Regardless of configuration (DIMM or PCIe), a program interacts with the Sieve device through the Sieve API, which supports calls to transpose a conventional database into the format needed for column-wise access (this can be stored for later use and is thus a one-time cost); load a database into the Sieve device; and make k-mer queries. The API implementation requires a user-level library and an associated kernel module or driver to interface to the Sieve hardware. The exact API and implementation are a subject to future work. K-mer databases are relatively stable over time, so once a database is loaded into the Sieve device, it can be used for long periods of time, until the user wishes to change a database. The same databases are often standard within the genomics community, high reuse can be expected to amortize the cost of database loading.

D. K-mer to Subarray Mapping

Without an appropriate mapping scheme, each query needs to be broadcast across all regions of the accelerator. A naïve mapping scheme would involve looking up an index table that maps queries to banks (Type-1) or subarrays (Type-2/3). Such a scheme would quickly stop scaling, as the size of such an index table increases exponentially with the length of a k-mer. We design an efficient and a scalable indexing scheme, wherein the size of the index table scales linearly with the main memory capacity rather than the length of a k-mer. More specifically, the reference k-mers in each subarray are sorted alphanumerically from left to right, and then each entry in our index table maintains an 8-byte subarray ID along with the integer values of the first and the last k-mers at the respective subarray (identified by the index). Upon receiving a matching request, Sieve first converts the query k-mer to its integer representation, and consults the index table to select the bank/subarray that contains a match. While Type-2/3 exploit different levels of parallelism, they share the same indexing scheme, i.e., if Type-2 only provides the bank address to our indexing scheme, a query needs to be checked against every subarray in that bank. The size of the index table stays well under 2 MB even for Type-2/3 with 500 GB of capacity, which is reasonable for a dedicated bioinformatics workstation.

E. Sieve: Putting it all together

For Type-2/3, the host reads the input query sequences and extracts k-mer patterns. For each k-mer, the k-mer to subarray index table is consulted to locate the destination subarray, and a k-mer request is made, as described in Section IV-C. A number of k-mer requests that need to be sent to the same subarray is grouped into one "batch". The exact number of k-mer requests per batch is equal to the number of query k-mers in a pattern group (64 in our example). These query batches are placed in a buffer, ready to be shipped to the PCIe device buffer by DMA. PCIe bundles several such batches into one PCIe packet (also described in Section IV-C) sent to the Sieve device. Sieve dispatches each batch of query k-mers to the destination subarray, and replaces an already processed query k-mer batch with a new (to-be-processed) batch.

Individual k-mer requests in the same batch potentially complete at different times as (1) they get issued out-of-order (as soon as their bank/subarray becomes available), and (2) each request may involve checking a different number of rows. Thus, response packets may arrive out-of-order at the host, where their sequence IDs and payloads are examined, as part of a post-processing step. Upon completion of all k-mer requests for a given sequence, the accumulated payloads are fed into a classification step, as illustrated in Figure 2. Note that there is no additional reordering step required at the host end as the accumulated payloads are typically used to build a histogram of taxons for a given DNA sequence.

TABLE I: Workstation Configuration

CPU Model	Intel(R) Xeon(R) E5-2658 v4
Core/ Thread/ Frequency	14/ 24/ 2.30 - 2.80 (GHz)
L1 (KB)/L2 (KB)/L3 (MB) $	32 / 256 / 35
Main Memory	DDR4-2400MHz
Memory Organization	32GB / 2 Channels / 2 Ranks
GPU Model	Pascal NVIDIA Titan X

TABLE II: Query Sequence Summary

Query files	# Sequences	Seq Length	# k-mer
HiSeq_Accuracy.fa (HA)	1.0e4 sequences	92 bases	6.2e4 k-mers
MiSeq_Accuracy.fa (MA)	1.0e4 sequences	157 bases	1.27e6 k-mers
simBA5_Accuracy.fa (SA)	1.0e4 sequences	100 bases	7.0e5 k-mers
HiSeq_Timing.fa (HT)	1.0e8 sequences	92 bases	6.2e8 k-mers
MiSeq_Timing.fa (MT)	1.0e8 sequences	157 bases	1.27e10 k-mers
simBA5_Timing.fa (ST)	1.0e8 sequences	100 bases	7.0e9 k-mers

V. Methodology

Workloads. We use Kraken2 [36] and CLARK [6] for the CPU baseline, and cuCLARK [37] for the GPU baseline. We use MiniKraken_4GB (4GB), MiniKraken_8GB (8GB), NCBI Bacteria (2785 genomes 6.24GB). The query sequences are summarized in Table II, and K is set to 31.

Baseline Performance Modeling. We report our workstation configurations in Table I. The GPU baseline is idealized because (1) the energy and latency of data transfer from host to GPU are not included, and (2) the on-board memory is assumed to always be large enough to avoid running each query multiple times. The baseline DRAM energy consumption is estimated by feeding memory traces associated with k-mer matching functions, obtained using Hopscotch [38], to DRAMSim2 configured to match our workstation. The CPU energy is measured using the Intel PMC-power tool [39], then scaled down by 30% to exclude the interference from other system components, and the GPU energy is measured using NVIDIA Visual Profiler [40] as it is performed in [41] to characterise the multi-GPU inference server energy efficiency and scaled down by 50% to exclude energy spent on cooling and other operations, consistent with the methodology from DRISA [19].

Circuit-level SPICE Validation. Of all the Sieve components, only the Matchers are in direct contact with the sense amplifiers' BLs. In the presence of the Matcher circuit, the load capacitance on the BL is increased. We use SPICE simulations to confirm that Sieve works reliably. The sense amplifier and matcher circuits are implemented using 45nm PTM transistor models [42]. Because of the relatively small input capacitance of the matcher circuit ($\sim$0.2 pf), in comparison with the BL capacitance ($\sim$22pf), the matcher has a negligible effect on the operation of the sense amplifiers.We find that, after the row activation and when the BL voltage is at a safe level to read, the result of the matcher is ready after less than 1 ns. To validate correct operation of links in Type-2, we use our DRAM circuit model to simulate transfer of data between local row buffers of two adjacent subarrays. In both simulations, the initial charge of the cell is varied across different values to consider the effect of DRAM cell charge variations. Even in the worst case, the matcher and the link between two subarrays cause no bit flips or distortions.

Energy, Area, and Latency Modeling. We estimate the power and latency overhead of each Sieve component using FreePDK45 [43]. Further, we use OpenRAM [44] to model and synthesize the SRAM buffer in Type-1. We use scaling factors from Stillmaker, et al. [45] to scale down results to the 22nm technology node, and use the planar DRAM area model proposed by Park, et al. [46] to estimate area overhead.

Modeling Sieve. We assume a pipelined implementation of Sieve, where the host (CPU) performs pre-processing (k-mer generation, driver invocation, and PCIe transfer) and post-processing (accumulation of response payloads for genome sequence classification), while Sieve is responsible for k-mer matching. Our analysis confirms that the latency of this pipeline is limited by k-mer processing on Sieve. In particular, k-mer matching on Sieve is either comparable to (for Type-3) or slower than (for Types-1/2) both pre- and post-processing steps on the CPU, so the CPU is always able to send enough k-mer requests to Sieve to keep it fully utilized.

We model the pre- and post-processing steps using the baseline CPU described in Table I. We treat the classification step as a separate pipeline by itself, as (1) the algorithm differs for each application, and (2) it is independent of k-mer matching, which is the primary focus of this work. Thus, we forgo modeling the effort required for genome classification and other post k-mer processing. For modeling the k-mer matching itself, we use a trace-driven, in-house simulator with a custom DRAMSim2-based front-end. The simulator also models PCIe communication overhead, using standard PCIe parameters [47]. We use a Micron DDR4 chip (DDR4_4Gb_8B_x16) as the building block for Sieve. DRAM parameters are extracted from the same datasheet and modified to account for the estimated latency and energy overhead of matchers, ETM, column finder, and segment finder.

VI. Experimental Results

A. Energy, Latency, and Area Estimation

Energy Evaluation. Table III summarizes the dynamic energy and static power of each Sieve component. Type-3 incurs additional power consumption for each DRAM row activation. However, using formula 10a from Micron's technical documentation [35], we find that Sieve consumes only 6% more energy for each row activation than a regular DRAM, because the area and the load of the extra transistors we introduce is so small compared to the sense amplifier and the bitline drivers. We further break down this energy overhead to understand the effect of the different Sieve components. We find that the Matcher Array (MA) and the ETM dominate the energy consumption, capturing 78.9% and 15.8% of the 6% energy overhead incurred by Sieve, with the energy spent by the Segment Finder and the Column Finder being negligible (less than 5%). Type-1 adds no overhead on top of the regular DRAM row activation, because no modification is made to the row buffer, and it is less energy-intensive than Type-2/3.

Latency Evaluation. Table III shows the latency of each Sieve component. For Type-1, we assume that (1) accessing

TABLE III: Sieve Components Energy and Latency Analysis

Component	Dynamic Energy (pJ)	Static Power (uW)	Latency (ns)
(T1) 64-bit MA	0.867	1.4592	0.353
(T1) QR, SkBR, StBR	1.92	5.28	0.154
(T1) SRAM Buffer	5.12	4.445	0.177
(T2/3) 8192-bit MA	181.683	0.289	0.535
(T2/3) ETM Segment	73.5	56.185	43.653
(T2/3) Segment Finder	2.44	0.294	0.362
(T2/3) Column Finder	20.69	28.16	0.152

the SRAM buffer and the Query Register can be overlapped entirely with a column read command ($\sim$15 ns) that retrieves a batch of reference bits, and (2) although the pattern matching and register checking are on the critical path, they add negligible overhead ($\sim$0.5 ns) to the DRAM row cycle ($\sim$50 ns). For Type-2/3, each ETM segment (256 OR gates) meets the timing requirement of completing its operation within one DRAM row cycle. Further, since the segment and column finders are composed of simple shifters, their latency of operation is well within one DRAM cycle (0.625 ns).

Area Evaluation. To estimate area the overhead of Sieve, we use the model proposed by Park et al. [46]. We adopt the DRAM sense amplifier layout described by Song, et al. [48] and a patent from Micron [49] for a conventional $4F^2$ DRAM layout. The short side and long side of the sense amplifier are 6F and 90F, respectively. In Type-2/3, for the accommodation of the matcher, ETM, segment, and column finder circuits in the local row buffer, we add 340F in total on the long side of the local sense amplifiers. For Type-2, an extra 60F in long side is added to each sense amplifier for considering the area overhead of the links between the subarrays.

The area overheads for Type-2 with 1, 64, and 128 compute buffers (CB) are 1.03%, 6.3% and 10.75%, for an 8-bank DRAM chip. In Type-3, each local sense amplifier is enhanced with k-mer matching logic, and for enabling subarray parallelism, a row-address latch is added to each subarray [28], resulting in 10.90% area overhead. For Type-1, all components are added to the center strip of our DRAM model. The SRAM buffer of 8 Kbits (128 Rows X 64 Bits) and matching circuit in each bank increase the area by 2.4% and 0.08%, individually.

B. Kernel Performance Improvement

Comparison Against Row-major In-Situ Accelerators. We simulate an ideal row-major baseline which mimics prior proposals [19], [20], [29] (Row_Major in Figure 13), and an improved row-major accelerator based on ComputeDRAM [31]. We measure their speedup over the CPU baseline. We also implement Sieve without ETM (Col-major).

We make the following assumptions for the Row-major, ComputeDRAM-based, and Col-major accelerators. First, their latency for locating and transferring payloads is assumed to be similar to that of Sieve. Second, both architectures are configured to be the same capacity with the same subarray-level parallelism. Third, they share the same indexing scheme. Fourth, we assume that ComputeDRAM has a much shorter Triple-row Activation latency due to the fact that it issues memory commands in rapid succession.

Figure 13 shows the results from this experiment. The convention for the workloads on the X-axis is kernel.query.size. The kernel is either Kraken2 or CLARK, the query files are listed in Table II, the sizes are 4GB, 8GB, and NCBI Bacterial reference (6.24GB). We make the following observations.

First, row-major perform similarly to column-major without ETM (slightly worse), but for different reasons. Column-major must activate all the rows that store k-mer data (64 rows if k=32). Row-major and ComputeDRAM stop when it finds a hit, but requires $\sim$10X more writes to set up the comparison, as each query k-mer must be replicated across the length of the row. Second, ComputeDRAM is able to outperform both the row-major and column-major (without ETM) approaches, owing to its fast triple-row activation. Third, the column-major approach used in Sieve allows it to benefit from our ETM strategy (that provides an additional speedup of 5.2X to 7.2X), in contrast to both row-major and ComputeDRAM designs that lack such an opportunity. We conclude that the chief contribution of column-major layout is therefore 1) in enabling ETM and 2) in amortizing the setup cost across a pattern group of 64 writes. The row-major design performs slightly worse than Type-3 without ETM because, in the event of a k-mer mismatch, both designs on average open roughly the same number of rows (62 8192-bit rows), but the row-major design stops when it finds a hit. We note that, from our evaluation, real sequence datasets are typically characterized by low k-mer hit rates (around 1%), thus favoring Sieve designs.

Leveraging ComputeDRAM to build a column-major k-mer matching accelerator entails solving many challenges. If we populate the query section with the same query, we will need 128×64 write commands per query (630X more than Sieve). Populating the query section with different queries brings more challenges. For example, there could be more than one match, impacting our ability to design an efficient indexing scheme. We note that addressing these challenges while maintaining the performance, efficiency, and cost benefits of these approaches is the subject of future work.

Improvement Over CPU. Figure 14 shows the average speedup and energy savings. All results are normalized to CPU measurement. In this experiment, we constrain the memory capacity of all designs to 32 GB. For Type-2, we consider all possible numbers of compute buffers per bank and select the midpoint of 16 (T2.16CB). We present the performance of other Type-2 configurations in Section VI-B. For Type-3, we choose the best performer, which supports 8 concurrently working subarrays (T3.8SA). While clearly more energy-efficient, Type-1 offers limited speedup (1.01X to 3.8X) for 8 out of 9 benchmarks, showing that for many workloads, there is significant additional performance potential that can be tapped via an in-situ approach. However, we also point out that Type-1 is likely to outperform CPU/GPU as its memory capacity grows (more banks thus more parallelism and bandwidth), while the similar memory-capacity-proportional performance scaling is hard to achieve in a non-PIM traditional architecture [50], due to the memory wall. Type-3

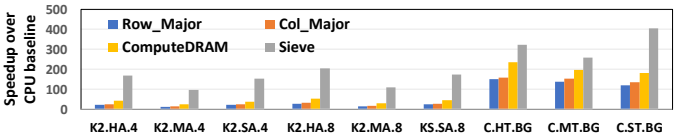

Fig. 13: **Row-major in-situ vs. Sieve Comparison.**

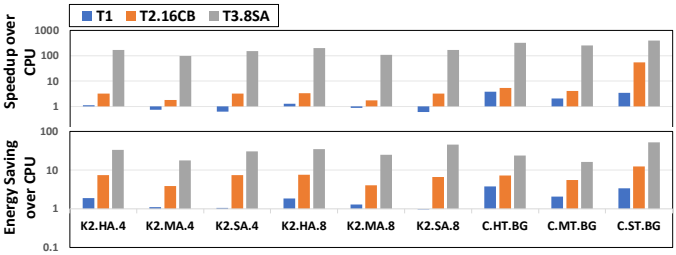

Fig. 14: **Comparison with CPU baseline.**

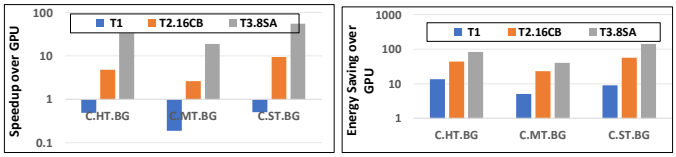

Fig. 15: **Comparison with GPU baseline.**

designs offer a speedup and an energy savings of as much as 404.48X and 55.89X respectively, over the CPU baseline. Note that this is in comparison to a Type-2 design that offers a speedup of 55.49X and an energy reduction of 28.11X over the CPU baseline, clearly showcasing the substantial benefits that can be realized by exploiting finer-grained parallelism at the subarray-level. We also find that Sieve is sensitive to the characteristics of the application. For example, the C.MT.BG benchmark performs worse than C.ST.BG benchmark as the number of k-mer matches for C.MT.BG is 3.28X higher than C.ST.BG benchmark, resulting in more row activations, increasing the overall query turnaround time and energy. Furthermore, recall from Section IV that our early termination mechanism interrupts row activations as soon as we detect a mismatch, minimizing the overall turnaround time and energy consumption for workloads with fewer k-mer matches.

Improvement Over GPU. Figure 15 shows the speedup and energy savings of various Sieve designs (32 GB) over the GPU baselines. Type-1 is 3X to 5X slower than the GPU but more energy efficient, and Type-2 is only modestly faster (2.59x to 9.43x). However, as the memory capacity of Sieve and dataset size increase, Type-1/2 are likely to outperform the GPU unless GPU memory capacity scales as fast, because all reference datasets can fit onto Sieve, avoiding the repetitive data transfer from host memory to GPU board. Type-3 dramatically outperforms the GPU, because it leverages subarray-level parallelism. Type-3 offers speedups of 33.13X–55.0X and energy savings of 83.77X–141.15x.

Effect of Increased DRAM Bandwidth. Simply increasing bandwidth to DRAM in the CPU and GPU baselines is not sufficient to address the performance bottleneck in k-mer matching, because we find that it is not bottlenecked by bandwidth. While it is memory-intensive (high percentage of loads in the ROB), memory bandwidth is underutilized

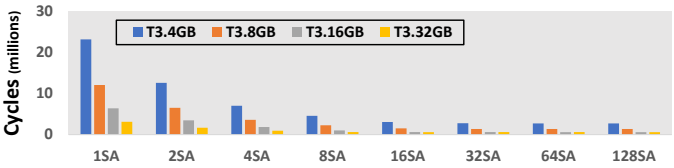

Fig. 16: **Average cycles spent to process CPU benchmarks.**

because each MSHR is unable to serve multiple loads and the available MSHRs are quickly depleted, stalling subsequent loads in the ROB and preventing the bandwidth from being fully saturated. Even if we overprovision those Broadwell cores with enough MSHRs to sustain all outstanding memory accesses, and all loads are served concurrently with a memory latency of 40 ns to reach the same level of throughput as Type-3, the workstation has to be equipped with over 215 cores, not only resulting in a substantial increase in power consumption, but a considerable wastage in DRAM bandwidth as only a small portion of the retrieved cache line is useful. cuCLARK is highly optimized, so we suspect that GPUs are constrained by similar bottlenecks as CPUs, although we have not yet pinpointed the exact set of microarchitectural structures.

C. Sensitivity Analysis

Number of Subarrays per Bank. We analyze the impact of subarray-level parallelism on performance and energy by comparing various Type-3 design configurations (see Figure 16) at different memory capacities and number of subarrays per bank. The results are averaged across all benchmarks. Supporting all subarrays performing k-mer matching simultaneously without increasing the area overhead significantly is not yet feasible, due to power delivery constraints. However, for this experiment, we assume this is not an issue. In any case, although Sieve's k-mer matching throughput increases with more concurrent subarrays, the speedup plateaus after 8 subarrays—probably because most bank-access conflicts can be resolved by a small number of subarrays [28].

Number of Compute Buffers. We explore the performance-area tradeoff of Type-2 designs, by varying the number of compute buffers (shown in Figure 17). For reference, we include Type-1 (the left-most bar T1) and Type-3 (the right-most bar T3.1SA) designs without subarray-level parallelism. The middle eight bars represent Type-2 with 1-128 compute buffers per bank. We make the following observations. First, Type-2 with one compute buffer is faster than Type-1 (1.39X to 1.94X) but not by a large margin. For each row activation, in the worst case, Type-1 has to burst read 128 batches to the matchers, which is similar to T2.1CB where the opened row needs to "hop" across 128 subarrays to reach the compute buffer. Since the hop delay (∼4ns) is faster than a burst latency (tCCD: 5∼7ns), and both design are equipped with some forms of ETM, T2.1CB is likely to spend less time on data movement than Type-1 in the average case. However, the chain activation of sense amplifiers in Type-2, which relays the row to the compute buffer, consumes significant energy, making Type-2 with sparse compute buffers less energy efficient. Second,

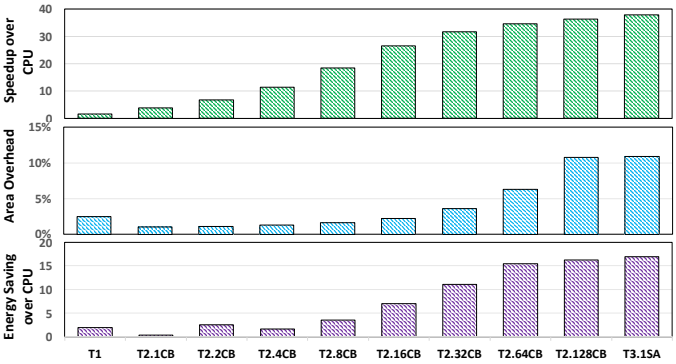

Fig. 17: **The effect of varying the number of compute buffers. T = Type, #CB = number of compute buffers.**

generally speaking, increasing the number of compute buffers per bank also increases the speed and energy efficiency of Type-2. As we have explained previously, adding more compute buffers reduces the activation of sense amplifiers, which in turn reduces the delay and energy consumption. Third, the area overhead scales with the number of compute buffers per bank. Finally, the speedup and energy reduction of T2.128CB slightly trails behind those of T3.1SA, because T2.128CB still requires one hop per row activation. However, Type-3 also has a higher area overhead than T2.128CB for enabling subarray-level parallelism.

ETM. To simulate the adversarial case where every query k-mer has a match, we turn ETM off in Type-2/3, and measure the speedup and energy reduction over CPU/GPU baselines (averaged across all benchmarks). Type-2/3 without ETM are still 1.34x–155.37x faster and 4.15x–36.17x more energy efficient than CPU, and 1.3X–9.54X faster and 6.60X–18.43X more energy efficient than GPU.

PCIe Overhead. We use PCIe 4.0 x16 in our simulation. Overall, PCIe adds 4.6% to 6.7% communication overhead to the ideal case where k-mer matching requests are dispatched to the destination bank/subarray as soon as they arrive, and returned to the host when they complete.

VII. Related Work

In this section we discuss previous work that shares similar interests concerning Sieve. The concept of PIM dates back to the 70s [51]. Since then, there have been many proposals integrating heavy logic into 2D planar DRAM dies [52]–[56]. These early efforts largely remain at their inception stage due to the challenges of fabricating logic using the DRAM process. Recently, the 3D-stacked technology, which takes a more practical approach by placing a separate logic die underneath the DRAM dies, revitalizes the interests in PIM research. To fully exploit the benefit of 3D-stacked architectures, many domain specific accelerators for graph processing [50], [57], pointer chasing [58], and data analytics [59] have been proposed. We plan to evaluate Sieve in 3D-stacked context as future work.

Non-DRAM-based In-situ Accelerators. NVM- and SRAM-based in-situ accelerators such as Pinatubo [60] and Compute Caches [61] have been proposed, but we choose DRAM for its maturity and availability, which can lead to quicker development and deployment cycles. Furthermore, SRAM generally has a lower capacity than that of DRAM, a smaller number of subarrays, and shorter row buffers. We plan to evaluate NVM-based Sieve in future work.

PIM-based Genomics Accelerators. Recently, PIM has been explored for several algorithm-specific PIM architectures for genomics. For example, GenCache [16] modifies commodity SRAM cache with algorithm-specific operators, achieving energy reduction and speedup for DNA sequence aligners. Medal [17] leverages commodity Load-Reduced Dual-Inline Memory Module (LRDIMM) and augments its data buffers with custom logic to exploit additional bandwidth and parallelism for DNA seeding. Radar [18] provides a high scalability solution for BLAST by mapping seeding and seed-extension onto dense 3D non-volatile memory. However, these efforts are not ideal for k-mer matching. GenCache has hardwired logic in SRAM to compute Shifted Hamming Distance and Myer's Levenshtein Distance, which are not used for k-mer matching. Medal is highly optimized for FM-index based DNA seeding, which relies on different data structures (suffix arrays, accumulative count arrays, occurrence arrays) than those in k-mer matching (associative data structures such as dictionaries). Radar binds seed-extension, a stage irrelevant to k-mer matching, with seeding to maximize speedup.

PIM-based Genomics Accelerators. PIM has been explored for several algorithm-specific architectures for genomics. For example, GenCache [16] is an SRAM-based accelerator for DNA sequence alignment. Medal [17] augments the data buffers of commodity DIMM to exploit additional bandwidth and parallelism for DNA seeding. Radar [18] provides a high-scalability solution for BLAST by mapping seeding and seed-extension onto dense 3D NVM. These efforts rely on domain-specific knowledge to achieve maximal speedup for specific algorithms that are not applicable to k-mer matching, but are complementary to Sieve.

VIII. Conclusions

In this work, we identify k-mer matching as a bottleneck stage in many genomics pipelines, due to its memory-intensive nature. We propose Sieve, a set of DRAM-based in-memory architectures to accelerate k-mer matching, by storing reference k-mer patterns along the bitlines and enhancing row buffers with a minimal set of Boolean logic for k-mer matching. We optimize Sieve with an Early Termination Mechanism. Type-1 offers limited benefit over CPUs and GPUs. Type-2 offers extensive speedups over CPUs (3.74x to 76.62x) but only modest benefit over GPUs (1.33x to 12.97x). Type-3 offers compelling benefits over both, with speedups and energy savings over the CPU of as much as 389.49X and 93.97X respectively; and 6.05x and 68.74x over the GPU.

IX. Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and suggestions. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor

Research Corporation (SRC) program, sponsored by MARCO and DARPA.

REFERENCES

[1] D. Navarro-Gomez, J. Leipzig, L. Shen, M. Lott, A. P. Stassen, D. C. Wallace, J. L. Wiggs, M. J. Falk, M. van Oven, and X. Gai, "PhyMer: a novel alignment-free and reference-independent mitochondrial haplogroup classifier," *Bioinformatics*, vol. 31, no. 8, 2015.

[2] Y. Li, T. B. Heavican, N. N. Vellichirammal, J. Iqbal, and C. Guda, "ChimeRScope: a novel alignment-free algorithm for fusion transcript prediction using paired-end RNA-Seq data.," *Nucleic Acids Res.*, 2017.

[3] D. E. Wood and S. L. Salzberg, "Kraken: ultrafast metagenomic sequence classification using exact alignments," *Genome Biology*, vol. 15, 2014.

[4] F. Breitwieser, D. Baker, and S. L. Salzberg, "KrakenUniq: confident and fast metagenomics classification using unique k-mer counts," *Genome Biology*, vol. 19, 2018.

[5] R. A. Edwards, R. Olson, T. Disz, G. D. Pusch, V. Vonstein, R. Stevens, and R. Overbeek, "Real time metagenomics: using k-mers to annotate metagenomes.," *Bioinformatics*, vol. 28, no. 24, 2012.

[6] T. J. C. Rachid Ounit, Steve Wanamaker and S. Lonardi, "Scalable metagenomic taxonomy classification using a reference genome database," *BMC Genomics*, vol. 16, 2015.

[7] S. K. Ames, D. A. Hysom, S. N. Gardner, G. S. Lloyd, M. B. Gokhale, and J. E. Allen, "Scalable metagenomic taxonomy classification using a reference genome database," *Bioinformatics*, vol. 29, no. 18, 2013.

[8] S. S. Minot, N. Krumm, and N. B. Greenfield, "One Codex: A Sensitive and Accurate Data Platform for Genomic Microbial Identification," *BioRxiv*, 2015.

[9] I. K. J. Anuj Gupta and L. Rishishwar, "stringMLST: a fast k-mer based tool for multilocus sequence typing," *Bioinformatics*, 2017.

[10] R. A. Vialle, F. d. Oliveira Pedrosa, V. A. Weiss, D. Guizelini, J. H. Tibaes, J. N. Marchaukoski, E. M. de Souza, and R. T. Raittz, "RAFTS3: Rapid Alignment-Free Tool for Sequence Similarity Search," *bioRxiv*, 2016.

[11] "Dna sequencing costs: Data."

[12] G. V. RESEARCH, "Metagenomics market size, share & trends analysis report by product (sequencing & data analytics), by technology (sequencing, function), by application (environmental), and segment forecasts, 2018 - 2025." GRAND VIEW RESEARCH, 2017.

[13] T. S. Rosing, N. Moshiri, and R. Knight, "Acceleration of bioinformatics workloads," *DARPA ERI Summit*, Jul 2020.

[14] C. L. Ventola, "The antibiotic resistance crisis: part 1: causes and threats.," *P & T : a peer-reviewed journal for formulary management*, vol. 40, no. 4, pp. 277–83, 2015.

[15] S. Altschul, W. Gish, W. Miller, E. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[16] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment," in *MICRO*, 2019.

[17] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, , and Y. Xie, "MEDAL: Scalable DIMM based Near Data Processing Accelerator for DNA Seeding Algorithm," in *MICRO*, 2019.

[18] W. Huangfu, S. Li, X. Hu, and Y. Xie, "RADAR: A 3D-reRAM Based DNA Alignment Accelerator Architecture," in *DAC*, 2018.

[19] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.

[20] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.

[21] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 556–569, 2020.

[22] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, 1995.

[23] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insights from a MICRO-46 Workshop," *MICRO*, 2014.

[24] R. Sharifi and Z. Navabi, "Online Profiling for Cluster-Specific Variable Rate Refreshing in High-Density DRAM Systems," *ETS*, 2017.

[25] J. A. Andrzej Zielezinski, Susana Vinga and W. M. Karlowski, "Alignment-free sequence comparison: benefits, applications, and tools," *Genome Biology*, vol. 18, 2017.

[26] K. L. A. Stinus Lindgreen and P. P. Gardner, "An evaluation of the accuracy and speed of metagenome analysis tools," *Scientific Reports*, vol. 6, 2016.

[27] S. Flygare, K. Simmon, C. Miller, Y. Qiao, B. Kennedy, T. Di Sera, E. H. Graf, K. D. Tardif, A. Kapusta, S. Rynearson, *et al.*, "Taxonomer: an interactive metagenomics analysis portal for universal pathogen detection and host mRNA expression profiling," *Genome Biology*, vol. 17, 2016.

[28] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *ISCA*, 2012.

[29] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.

[30] A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain, "FPGA-accelerated seed generation in mercury BLASTP," in *FCCM*, 2007.

[31] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 100–113, Association for Computing Machinery, 2019.

[32] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017.

[33] E. Fernandez, W. Najjar, and S. Lonardi, "String Matching in Hardware Using the FM-Index," in *FCMM*, 2011.

[34] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *HPCA*, 2016.

[35] Micron, "Tn-40-07: Calculating memory power for ddr4 sdram introduction." https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf, 2020.

[36] B. L. Derrick E Wood, Jennifer Lu, "Improved metagenomic analysis with kraken 2," *Genome Biology*, vol. 20, 2019.

[37] M. A. Kobus Robin, Hundt Christian and B. Schmidt, "Accelerating metagenomic read classification on CUDA-enabled GPUs," *BMC Bioinformatics*, 2009.

[38] A. Ahmed and K. Skadron, "Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation," in *MEMSYS*, 2019.

[39] "Intel Performance Counter Monitor." https://github.com/opcm/pcm.

[40] "NVIDIA Visual Profiler." https://developer.nvidia.com/nvidia-visual-profiler.

[41] A. Jahanshahi, H. Sabzi, C. Lau, and D. Wong, "GPU-NEST: Characterizing Energy Efficiency of Multi-GPU Inference Servers," *CAL*, 2020.

[42] "Predictive Technology Model (PTM)." http://ptm.asu.edu/.

[43] "FreePDK45: 45nm variant of the FreePDK Process Design Kit." https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[44] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *ICCAD*, 2016.

[45] A. Stillmaker, Z. Xiao, and B. M. Baas, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," 2012.

[46] J. B. Park, W. R. Davis, and P. D. Franzon, "3D-DATE: A Circuit-Level Three-Dimensional DRAM Area, Timing, and Energy Model," *IEEE Transactions on Circuits and Systems*, 2019.

[47] "PCI-E Specification." https://pcisig.com/specifications.

[48] K. Song, J. Kim, J. Yoon, S. Kim, H. Kim, H. Chung, H. Kim, K. Kim, H. Park, H. C. Kang, N. Tak, D. Park, W. Kim, Y. Lee, Y. C. Oh, G. Jin, J. Yoo, D. Park, K. Oh, C. Kim, and Y. Jun, "A 31 ns Random Cycle VCAT-Based 4F ^{2} DRAM With Manufacturability and Enhanced Cell Efficiency," *IEEE Journal of Solid-State Circuits*, 2010.

[49] Micron, "4 f2 folded bit line dram cell structure having buried bit and word lines." https://patents.google.com/patent/US6689660B1/en, 2020.

[50] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[51] H. S. Stone, "A Logic-in-Memory Computer," *IEEE Transactions on Computers*, vol. C-19, no. 1, pp. 73–78, 1970.

[52] Yi Kang, Wei Huang, Seung-Moon Yoo, D. Keen, Zhenzhou Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: toward an advanced intelligent memory system," in *ICCD*, 1999.

[53] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *MICRO*, 1997.
[54] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, Jaewook Shin, and Joonseok Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *SC*, 1999.
[55] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A Memory-SIMD Hybrid And Its Application To DSP," in *CICC*, 1992.
[56] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the terasys massively parallel pim array," *Computer*, 1995.
[57] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *HPCA*, 2018.
[58] K. Hsieh, S. M. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," *ICCD*, 2016.
[59] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *ISCA*, 2017.
[60] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016.
[61] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *HPCA*, 2017.