NvWa: Enhancing Sequence Alignment Accelerator Throughput via Hardware Scheduling

Yewen Li[1,2], Xueqi Li✉[1], Ruihao Gao[1,2], Wanqi Liu[1,2], Guangming Tan[1,2]
[1] State Key Lab of Processors, ICT, CAS
[2] University of Chinese Academy of Sciences, Beijing, China
Email: {liyewen19s, lixueqi, gaoruihao20s, liuwanqi18z, tgm}@ict.ac.cn

***Abstract*—Sequence alignment is the most time-consuming step in the genome analysis pipeline. Since sequence alignment generally follows the *seed-and-extension* paradigm, prior proposed hardware accelerators either opt to accelerate the *seeding* phase or the *seed-extension* phase. However, the diversity of each sequence in the alignment workflow leads to the pipeline stall or bubbles, which finally results in a decreased throughput for the end-to-end sequence alignment.**

In this paper, we propose NvWa, which is a hardware scheduling accelerator for sequence alignment. To solve the diversity problem, we propose three novel scheduling mechanisms and corresponding architecture, which target the *seeding* phase, the *seed-extension* phase, and the interaction between the two phases, respectively. For the *seeding* phase, we propose a Seeding Scheduler to schedule all idle seeding units in only one cycle. For the *seed-extension* phase, the Extension Scheduler can achieve both lower latency and higher parallelism when facing seed-extension tasks with different scales. Between the two phases, an efficient Coordinator caches and dispatches seeding hits to optimal and sub-optimal seed-extension units. Furthermore, to avoid algorithmic obsolescence for the new sequence technologies, we propose a loosely coupled design, which decouples the data path and the control scheduling path. Experimental results show that NvWa can achieve 493×, 200×, 12.11×, 2.30× speedup and 14.21×, 5.60×, 4.34×, 5.85× energy reduction when compared with a 16-thread CPU baseline, an NVIDIA A100 GPU, and two state-of-the-art accelerators, respectively.

I. Introduction

Sequence alignment is the first essential and the most time-consuming step in genome data analysis [28], [66]. In sequence alignment (aka. read alignment), each read is matched to its possible locations in the reference genome. Generally, the sequence alignment follows the *seed-and-extend* paradigm and can be divided into two phases, *seeding* and *seed-extension* [36], [38]–[41]. The *seeding* phase performs exact matching to generate a set of hits that indicate the raw reads' possible alignment locations. The *seed-extension* phase adopts the approximate string-matching methods to extend the hits. During this process, the two phases follow the *producer-consumer* model, which means the *seeding* phase produces many candidates (hits), and they will be extended to longer sub-strings one by one in the *seed-extension* phase.

Previous work has designed efficient accelerators for the *seed-and-extend* paradigm since it occupies more than 80% of the time of the whole sequence alignment [6], [18], [66]. Several studies optimize the *seeding* phase [6], [10], [18], [21], [26], [33], [57], [60], [61], [64], [65]. MEDAL [32] and FindR [69] use device properties such as PIM to accelerate the original algorithm, while EXMA [33] and ERT [57] propose new seeding algorithms (e.g., MTL-index and Enumerated Radix Tree). Some other work accelerates the *seed-extension* phase [13], [15], [17], [20], [23], [24], [27], [29], [34], [42], [44], [47], [58], [60], [61]. SeedEx [24] and Darwin [60] are based on traditional dynamic programming algorithms (e.g., Needleman-Wunsch, Smith-Waterman, and banded Smith-Waterman), while GenAx [23] and GenASM [16] are based on novel approximate string matching algorithms (e.g., Automata and Bitap).

According to our observation, a significant **diversity problem** is haunting the throughput of existing designs (cf. Sec. III). The performance of previous designs may not be fully utilized due to system starvation or blocking. Plenty of work uses two different computing units instead of one for different phases since the *seeding* phase is memory-bound and the *seed-extension* phase is compute-bound. The design of the two different computing units and the characteristics of the input read jointly result in the diversity problem. We argue that multi-phase scheduling is a promising solution to this problem. However, scheduling is not trivial because of three challenges: **Seeding Termination Diversity** (Challenge-❶), **Extension Scales Diversity** (Challenge-❷), and **Hit Characteristics Diversity** (Challenge-❸).

Recent hardware designs have adopted different scheduling methods but do not solve these challenges well. For example, Darwin [60] utilized software API to control the dataflow and hardware execution. SeedEx [24] utilized a non-blocking producer-consumer buffer for communication between FPGA and host. ERT [57] was based on SeedEx and employed a queue to deal with variable tree traversal times, and scheduled walks from other reads in the *seeding* phase. However, the software API of Darwin suffered from hardware and software switching overhead. The buffer of SeedEx can only provide coarse-grained data control and cannot exploit the data characteristics. The queue in ERT ignored the inter-unit level diversity. Neither approach is good enough.

In this paper, we propose NvWa[1], which leverages multi-

✉ Corresponding Author

[1]NvWa is named after a goddess of mending the cracks in the sky in Chinese mythology, indicating our accelerator fills the gap between the *seeding* phase and the *seed-extension* phase.

phase scheduling mechanisms and architectures to enhance the throughput of sequence alignment accelerators.

For Challenge-❶, we focus on improving the resource utilization of seeding units and avoiding the starvation of the seed-extension unit. We propose a Seeding Scheduler for the *seeding* phase, which dynamically allocates unprocessed reads to idle seeding units. We achieve the scheduling of all idle units in only one cycle at 1GHz at the algorithmic and microarchitectural levels, respectively. For Challenge-❷, we propose an Extension Scheduler for the *seed-extension* phase. We first quantitatively analyze that existing designs can not satisfy the different task scales and then propose a resource allocation strategy to achieve lower latency and higher parallelism when dealing with tasks of different scales. And we further use this strategy to configure the number of processing elements (PEs) of the seed-extension units. For Challenge-❸, we design a Coordinator to address the inter-phase problem caused by the *seed-and-extend* paradigm. The Coordinator decreases the system's sensitivity to input reads distribution and leverages a greedy allocating algorithm and lightweight architecture to assign seeding hits to optimal or sub-optimal seed-extension units.

In addition, the computing units (i.e., the seeding units and the seed-extension units) of NvWa are faithful to the standard read alignment software, which allows us to have no loss of accuracy. Also, our designs (i.e., the Seeding Scheduler, the Extension Scheduler, and the Coordinator) are orthogonal to previous work since we focus on loosely coupled scheduling design rather than proposing a new design of computing units.

Our specific contributions are listed as follows:

- We identify an execution diversity problem that appears in the *seed-and-extend* paradigm, which limits the end-to-end sequence alignment throughput.
- We propose a high-throughput and efficient hardware-scheduling-based accelerator, i.e., NvWa, for the end-to-end sequence alignment pipeline. In addition, NvWa has no loss of accuracy and is orthogonal to previous designs.
- We introduce multi-phase scheduling mechanisms and microarchitecture implementation in NvWa. Specifically, We design the Seeding Scheduler, the Extension Scheduler, and the Coordinator, which target the *seeding* phase, the *seed-extension* phase, and the interaction, respectively.
- Our experimental evaluation shows that NvWa can provide 493×, 200×, 12.11×, 2.30× speedup and, 14.21×, 5.60×, 4.34×, 5.85× energy saving over a 16-thread BWA-MEM [38], GASAL2 [5], GenAx [23], and GenCache [49].

II. Background

A. Execution Pipeline

As shown in Fig. 1, the read alignment pipeline has the following four steps [7], [8], [37], [50], [55].

- **Step-❶: Find Seeds.** The read accepts a start position as input and extends forward and backward as long as possible using exact matching algorithms. After this step, we get some small read subsets called seeds, which are noted as Seed 0, Seed 1, Seed 2, and Seed 3 in Fig. 1.
- **Step-❷: Filter and Chain.** Short seeds are filtered out while seeds with close coordinates chain each other into longer seeds by introducing a few edit errors. In Fig. 1, Seed 1 is filtered out, and Seed 2 and Seed 3 are chained to Seed 2+3.
- **Step-❸: Seeds Extension.** Potential candidate seeds (Seed 0, Seed 2+3) are extended forward and backward using compute-intensive approximate matching algorithms.
- **Step-❹: Get Result.** We select the Extension 1 as the final result, which is the extension version of Seed 2+3 and has the highest score.

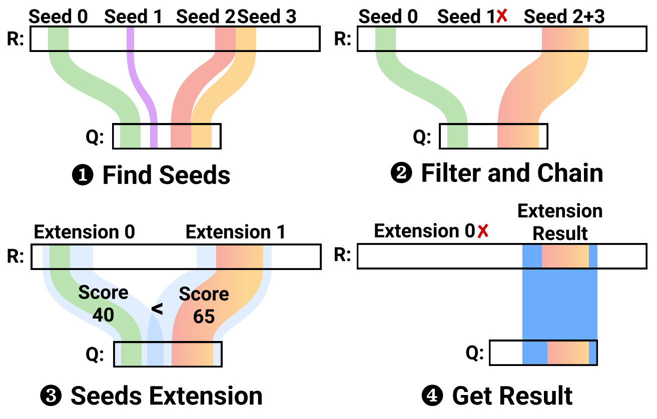

Fig. 1. Sequence alignment pipeline of a single read.

B. Widely Accepted Algorithms

Seeding Phase Algorithm. The FM-index search algorithm [22] and the Hash-based search algorithm [9] are widely used in this phase. The FM-index search algorithm realizes a fast search of candidate locations in the genome for short sequences by retrieving a BWT-based compression index structure for the reference genome. The drawback of this algorithm is the slow speed of matching due to frequent memory access. The Hash-based search algorithm scans the reference genome sequence and splits it into small fragments of length k, also known as k-mer, and builds a hash table by counting the occurrence of each k-mer. The exact matching process is to find the occurrence position of k-mers. The benefit of this method is the relatively regular memory access, and the drawback is its $O(4^k)$ memory consumption. Some other algorithms, such as ERT (Enumerated Radix Trees) [57] and D-SOFT [60], [61] are also used in this phase.

Seed-Extension Phase Algorithm. The Smith-Waterman algorithm [56] is commonly used in this phase since it can provide local-optimal alignment for biological scoring schemes. A typical scoring scheme has three parts: substitution matrix, open gap penalty, and extension gap penalty. The two sequences to be aligned operate in matrix-fill and trace-back two steps using the scoring schemes [60]. Several other algorithms, such as Bitap [16] and Automata [23], can also be used to perform this phase.

C. State-of-the-Art Accelerators

We present several typical studies relevant to the following papers, and more details about other state-of-the-art accelerators are presented in Sec. VII.

Darwin [60] and Darwin-WGA [61]. For the *seeding* phase, they leverage a Hash-based search algorithm [30] to alleviate the random access problem of the traditional FM-index search algorithm. For the *seed-extension* phase, Darwin and Darwin-WGA propose GACT based on the Smith-Waterman algorithm, which can use constant hardware resources to perform an arbitrary length matching.

GenAx [23] and GenCache [49]. The contribution of GenAx is mainly focused on the *seed-extension* phase. It improves Levenshtein Automata by proposing Silla, which can support the feature of string matching and supports arbitrary length matching. For the *seeding* phase, GenAx uses a similar approach as Darwin and Darwin-WGA. GenCache, which is based on GenAx, improves the throughput and reduces the bandwidth requirement using algorithm-hardware co-design.

III. Motivations and Challenges

A. Motivations

Diversity Problem in Seed-and-Extend Paradigm. Fig. 2(a) depicts the execution time breakdown of the *seeding* and *seed-extension* phase when running the standard software BWA-MEM [38] using massive reads sampled from the standard genome sequence[2]. The hardware platform and genome datasets will be described in Sec. V-B. Fig. 2(b) is the zoom-in on execution time breakdown for Read ID from 350 to 400 of Fig. 2(a).

For each read, it is evident that the proportion of the *seeding* and the *seed-extension* phase varies, and the total execution time is also different. If the running time of the *seeding* phase in a certain period is lower than the *seed-extension* phase, a large number of hits can not be processed, and the entire system is congested, which wastes the hardware acceleration efficiency of the *seeding* phase. Otherwise, if the execution time of the *seed-extension* phase in a certain period is lower than the *seeding* phase, the system is starved. The efficiency of the *seed-extension* phase is wasted too. These states affect resource utilization and downgrade performance.

The Need for Efficient Multi-Phase Hardware Schedulers. Fig. 3(a) shows an abstraction of the accelerator design used so far [23], [24], [57], [60], [61] to accelerate the *seed-and-extend* pipeline yet fails to solve the problem described above. In contrast, Fig. 3(b) illustrates that introducing scheduling mechanisms can efficiently solve the above problems.

[2]We profile BWA-MEM as it is the de facto standard 2nd generation read alignment software [14], [43], [46], [53], [66]. Other optimized versions, such as BWA-MEM2 [62], have similar statistical properties since their algorithms have not been fundamentally changed.

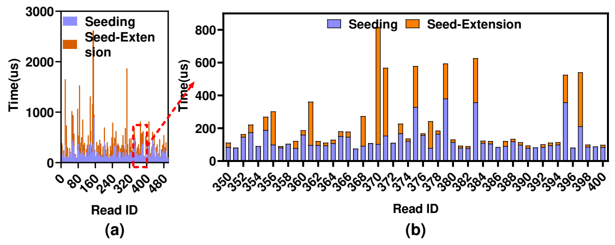

Fig. 2. Execution time breakdown of the seeding and seed-extension phases for 500 reads sampled from NA12878. (b) Zoom in on the execution time breakdown for Read ID from 350 to 400.

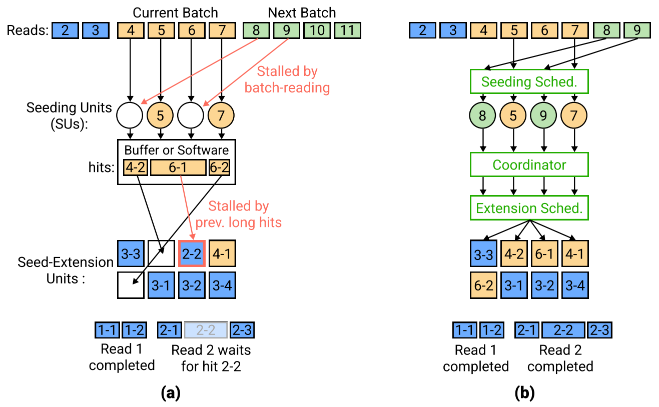

Fig. 3. The execution breakdown with or without scheduling for accelerating the *seed-and-extend* paradigm. (a) The traditional approach used so far [23], [24], [57], [60], [61]. (b) The traditional approach + multi-stage scheduling (the Seeding Scheduler, the Extension Scheduler, and the Coordinator).

- A coarse-grained Read-in-Batch strategy results in the seeding units (SUs) being idle. In contrast, when the Seeding Scheduler is integrated (in Fig. 3(b)), the input reads can be controlled at a fine-grained level. For example, SU 1 and SU 3 in Fig. 3(b) can load read 8 and read 9, which are not loaded in Fig. 3(a) since the current batch is not completely finished.
- The execution time of seed-extension units (EUs) is sensitive to the length of the input hits. This phenomenon causes that hits with different lengths can not be allocated to the most optimal computing resources. In contrast, fine-grained control over EUs is available when Extension Scheduler is introduced.
- Additionally, the Coordinator in Fig. 3(b) can evaluate the length of the hit and then assign the optimal computing unit for the hits with different lengths. For instance, Fig. 3(b) can allocate computing units for hit 6-1, which is blocked in Fig. 3(a) since hit 2-2 is not allocated the optimal computing unit.

B. Challenges

We summarize three design challenges for designing efficient multi-phase hardware schedulers as follows.

Challenge-❶: Seeding Termination Diversity. SUs are the data producers of the entire accelerator, and they must keep on generating seeds to avoid the starvation of EUs. The key challenge is to schedule the idle SUs efficiently when considering that the execution time of SUs is sensitive to the input reads.

Challenge-❷: Extension Scales Diversity. EUs are the consumers in the system, and they are required to be low latency and high parallelism to consume the hits generated during the *seeding* phase. As shown in [20], the length of hits extended in the *seed-extension* phase is variant significantly for each read. The key challenge is to avoid the under-utilization and workload imbalance of the EUs when the hardware design is specified in advance.

Challenge-❸: Hit Characteristics Diversity. Since the read alignment follows the *seed-and-extend* paradigm, all valid hits generated by SUs need to be executed by EUs. The key challenge is to cache and analyze them in a lightweight way since the number and characteristics of the hits generated by each SUs are varied.

IV. Architecture Design

A. Architecture Overview

Fig. 4 depicts the architecture overview of NvWa, which includes the following five parts and solves the three challenges mentioned above.

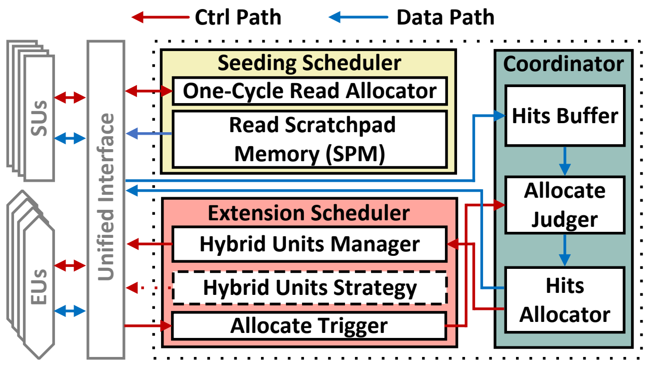

Fig. 4. Architecture overview of NvWa

Solving Challenge-❶. The Seeding Scheduler is targeted to realize the efficient execution of the SUs when faced with workloads that require different execution times and can not be predicted in advance (cf. Sec. IV-B). The One-Cycle Read Allocator avoids the competing problem of idle SUs and assigns unfinished workloads to idle SUs within one cycle, and the Read SPM is used to prefetch the reads that are to be processed, hiding the access latency of DRAM.

Solving Challenge-❷. The Extension Scheduler is designed to maximize the efficient execution of regular accesses and computations of EUs (cf. Sec. IV-C). The Allocate Trigger is responsible for checking the execution status of the EUs and deciding whether to send a scheduling request to the Coordinator based on the number of idle units. The Hybrid Units Strategy acts as a guideline to modify the EUs to provide low latency and high parallelism. The Hybrid Units Manager receives the scheduling results from the Hits Allocator and distributes them to the specified EUs.

Solving Challenge-❸. The Coordinator buffers the hits finished by the SUs and allocates them reasonably to the idle EUs (cf. Sec. IV-D). The Hits Buffer receives the results (hits) of SUs and sends the hits to the Hits Allocator for scheduling. The Allocate Judger receives scheduling requests from the Allocate Trigger. It activates the Hits Allocator to perform a round of hits scheduling. The Hits Allocator, the core component of the Coordinator, dispatches the data in the Hits Buffer to the optimal or near-optimal idle EUs with a low-latency greedy allocation architecture.

Other Components. The SUs and The EUs refer to prior accelerators that are faithful to the standard algorithm and can access the above three scheduling components through the Unified Interface. The implementation of SUs and EUs is described in Sec. V-A. The utility of the Unified Interface is discussed in Sec. VI.

B. Seeding Scheduler Design

Inefficiency of Current Scheduling Strategy. Previous work included basic schedulers but was not sufficiently optimized. Read-in-Batch is a typical approach adopted by state-of-the-art seeding accelerators such as GenAx [23] and ERT [57]. This solution is easy to implement but suffers from poor utilization of SUs when considering that the execution time of SUs is sensitive to input reads[3]. Fig. 5(a) illustrates the execution flow using this strategy to schedule four SUs. The system starts at cycle 0, and all SUs are idle. At cycle T_0, four SUs load four reads from memory and start to execute the *seeding* phase. SU 1 and SU 3 are done at cycle T_1, and SU 2 is done at cycle T_2. Since not all SUs are finished, we can not feed data for these idle SUs. The slowest SU 0 finishes at T_3, and all SUs start computing the next batch at $T_3 + 1$ when the reads are prefetched. From this toy example, we can observe that the Read-in-Batch strategy leads to idle SUs and does not provide a high resource utilization.

Our One-Cycle Scheduling Strategy. We introduce One-Cycle Read Allocator that correctly allocates to each idle SU within one cycle to solve this problem. The key idea of the One-Cycle Read Allocator is assigning a priority corresponding to the index for each computing unit. The idle computing unit with a smaller index has higher priority than the idle unit with a bigger index. Suppose we have N SUs and each of them has a status s_i, where s_i is 0 for idle, and s_i is 1 for busy. a_i is the allocated read index of data for each unit, and the global read index offset is g. Then, at each cycle, the updating formula of a_i and g is

$$a_i \leftarrow \begin{cases} g + 1 + \sum_{k=0}^{i-1} (1 - s_k) & \text{if } s_i = 0 \\ a_i & \text{if } s_i = 1 \end{cases}, \qquad (1)$$

[3]The execution time of the Hash-based search algorithm is as input-sensitive as that of the FM-index search algorithm. For example, the number of DRAM accesses for Darwin [60] is 2+P. 2 is the times to access the pointer table. P is the times to access the position table, which is a variable for different k-mers.

$$g \leftarrow g + \sum_{k=0}^{N-1} (1 - s_k)\,. \tag{2}$$

A toy example is given in Fig. 5(b). At cycle $T_1 + 2$, unit 1 and unit 2 will be allocated with read 4 and read 5. The read index allocated to unit 1 is 4, given that unit 0 is busy. Furthermore, unit 2 will be allocated with read 5, considering that unit 0 is idle. Using this strategy, we can allocate reads to multiple units in parallel. Compared to Fig. 5(a), this strategy can feed unprocessed reads to idle units at cycle T_1, cycle T_2, and cycle T_3.

Microarchitecture Design. Fig. 6 shows the microarchitecture of the One-Cycle Read Allocator. The masks shown in the upper left corner represent the hardware design of the priority described above. For example, unit 0 corresponds to a mask of 0000, and unit 3 corresponds to 1110. Our microarchitecture has five steps. ❶ Inverse *unit_status* and *unit_status[i]*. ❷ Get the number of 1's before *unit[i]* by taking a AND operation between *unit_mark_table[i]* and *unit_status*. ❸ Obtain the exact number of 1's using a PopCount Tree, the number of 1's indicates the number of units that need to be loaded a new read before *unit[i]*. This result indicates the local loading read index of the *unit[i]* when considering all the idle units at this cycle. ❹ Add this result with *read_offset* to get the global loading read index. ❺ Check if the *unit[i]* needs to load a new read using a Mux operation.

The latency of the design depends on the depth of the PopCount tree. In practice, the number of seeding units is from 64 to 512, and the depth of the tree is from 6 to 9, which makes the hardware latency requirements can be easily satisfied at 1 GHz.

Discussion of Intra-Unit Level Scheduling Strategy. The switching context of ERT [57] is a scheduling mechanism within each SU for hiding the latency of accessing DRAM. Compared with ERT, the One-Cycle Read Allocator eliminates the bubbles caused by DRAM latency at the intra-unit level and the ones caused by the diversity of execution time at the inter-unit level. The One-Cycle Read Allocator is used for inter-unit scheduling, and the switching context is used for intra-unit scheduling. To summarize, our One-Cycle Read Allocator and the switching context proposed in ERT are orthogonal, and they can work together for seeding engines.

C. Extension Scheduler Design

Inefficiency of Current Design. The execution pattern of the *seed-extension* phase is regular and compute-intensive, and the most widely known structure is the systolic array [45], [47], [60], [61], [67]. As shown in Fig. 7(a) and Fig. 7(b), the query sequence is divided into three blocks of GCG, CAA, and TGT, and the three bases of each block are placed in three PEs, respectively. The reference sequence is loaded into the PEs from the left. The PEs compute the alignment result, and the computation results are stored in the SRAM cache below. The final result is obtained using the trace-back logic unit when all blocks are finished[4].

Fig. 7(c) gives the execution flow of the matrix-fill step. The computation of *Block 0* needs to be completed first. In cycle 1, the first element G of *Block 0* compares with the first element G of the reference sequence. In cycle 9, the first element G of *Block 0* completes the comparison with the whole reference sequence. Since the third element G of *Block 0* is delayed by two cycles compared to the first element, the computation of the whole block is completed at cycle 11. The remaining two blocks are computed in the same form as *Block 0*, so the computation cycle to complete the whole process is the execution cycle of a single block multiplied by the number of blocks. The whole process consumes 33 cycles.

Based on the above discussion, let R be the length of the reference sequence, let Q be the length of the query sequence,

[4]Since the latency of the trace-back logic is constant for a specific query and reference, irrelevant to the number of PEs, it is not discussed here.

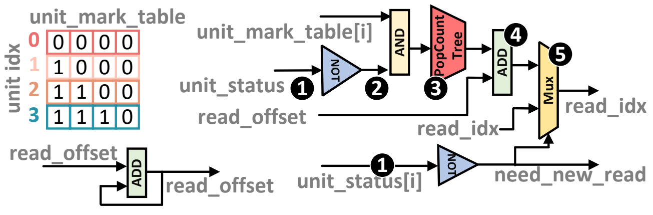

Fig. 6. The microarchitecture of the One-Cycle Read Allocator.

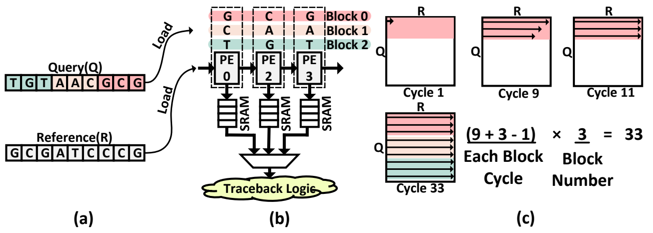

Fig. 7. A runtime example of the well-known systolic array [45], [47], [60], [61], [67]. (a) The query sequence and the reference sequence. (b) Common-used hardware design of the systolic array. (c) Execution flow of running (a) on (b).

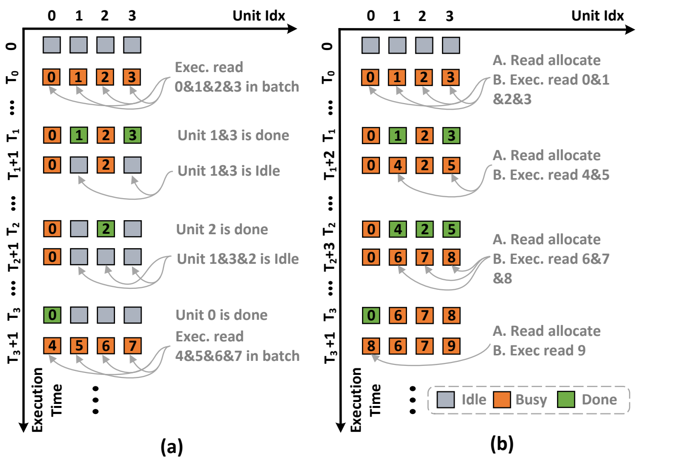

Fig. 5. Comparison of Read-in-Batch strategy and One-Cycle scheduling strategy. (a) Read-in-Batch strategy. (b) One-Cycle scheduling strategy.

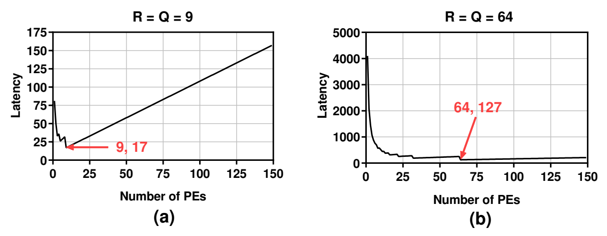

Fig. 8. Latency of systolic array with different number of PEs.

and let P be the number of PEs. The latency L of the matrix-fill process is

$$L = (R + P - 1) \times \left\lceil \frac{Q}{P} \right\rceil . \quad (3)$$

As an example, Fig. 8 presents the execution latency of the systolic array [45], [47], [60], [61], [67] with different numbers of PEs for the cases of sequence lengths of 9 and 64. We can get three key observations from this figure. (1) When the hit length and the number of PEs are close to each other, the computation has the shortest latency. (2) A short hit running on a large PE incurs high latency due to idle units, and a long hit running on a small PE also incurs high latency due to multiple iterations. (3) The latency of short hits running on small PEs and long hits running on large PEs is not much different from the shortest latency and can be chosen as the sub-optimal solution.

Hybrid Units Strategy. To balance latency and resource usage, we propose a Hybrid Units Strategy to handle the different scales of the task. Suppose Fig. 9(a) shows the distribution of hit lengths obtained by software execution, and here we divide the data distribution into four intervals. Fig. 9(b) shows that the conventional approach would use moderately sized units to handle all tasks. Here we take the number of PEs as 64 and take the number of units as four, and thus the total number of PEs is 256. Our proposed approach will design units with different numbers of PEs according to the distribution of hits length. As shown in Fig. 9(c), We design five PEs, two of which contain 16 PEs, and the remaining three contain 32, 64, and 128 PEs, respectively. The total number of PEs is still 256.

Fig. 9(d) presents the cycles required to execute the hits (20, 40, 10, 65, 127) for both uniform and hybrid units. Since only four units are available for the uniform units, only the first four hits can be computed first at cycle 1. The execution latency for different hits can be obtained by Formula 3. At cycle 74, the third unit completes the computation of the hit 10 with the shortest latency, and the remaining hit 127 is loaded at cycle 75. At cycle 84, the first unit completes the computation of hit 20. The subsequent process follows this procedure and takes 455 cycles to complete the computation of all hits. In contrast, our strategy has more units than the uniform units strategy, and all the hits can be loaded at once. Furthermore, different computing units are customized for different lengths of hits. The latency is universally lower than that of the uniform units. Similarly, it requires 257 cycles to complete the computation of all hits. Compared with the uniform unit strategy, our strategy makes a better trade-off between parallelism and latency with a reasonable scheduling mechanism (detailed in Sec. IV-D).

The above analysis is based on Fig. 9(b) using four units, each with 64 PEs. It can be achieved by assigning 51 PEs to each unit, then the total number of units in Fig. 9(b) is also five, and the PEs consumed are approximately the same as in Fig. 9(c) (i.e., 255 vs. 256). Based on the long execution latency problem in Formula 3, this allocating strategy still can not outperform our hybrid approach.

Guideline for Assigning Hybrid Units Based on Hits Distribution. Typically, we need to determine the number of hybrid EUs in each class based on a hit distribution and the total number of PEs. The hit distribution can be derived from a standard dataset or the average of multiple datasets. Empirically, the number of PEs for each class is preferentially determined as a power of two for design simplicity. The class of hybrid EUs is a hyperparameter, which is usually taken as four. We explore the design space for this parameter in Sec. V-E.

Suppose there are n classes of EUs and the number of PEs in the i-th class is p_i. For a given distribution of hit length, we divide the entire region into n intervals according to the value of p_i, and then the summation is performed within the interval, and the summation for each interval is noted as s_i. For the i-th interval, p_i is the optimal execution unit according to Formula 3. Let the total number of PEs be given as N and the number of EUs of the i-th class x_i, we can derive the following set of equations,

$$\begin{cases} \sum_{i=0}^{n} x_i \cdot p_i = N \\ x_0 : x_1 : \cdots : x_{n-1} = s_0 : s_1 : \cdots : s_{n-1} \\ x_i \in Z, i \in [0, n-1] \end{cases} . \quad (4)$$

By solving this set of equations, we can obtain the number of class i-th EUs as

$$x_i = \frac{s_i \cdot N}{\sum_{j=0}^{n-1} p_j \cdot s_j}, i \in [0, n-1]. \quad (5)$$

Discussion of Other State-of-the-Art Seed-Extension Accelerators. Although the analysis above is based on the systolic-array-based designs [60], [61], our strategy also applies to other state-of-the-art studies.

GenASM [16] adopts a fixed number of PEs to align input hits. The part exceeding the number of PEs will be computed serially by iteration. GenAx [23] uses a fixed number of PEs to align hits with different edit distances. The number of edit distances exceeding the number of PEs is computed by composing multiple tiles with a fixed number of PEs. Compared with those designs, our approach fine-grained tuning the scale of computing units according to the length of hits can reduce resource wastage and computing latency. In addition, our scheme is orthogonal to the above designs. SeedEx [24] can handle input data of arbitrary length, but there still has a trade-off between the execution band size and performance

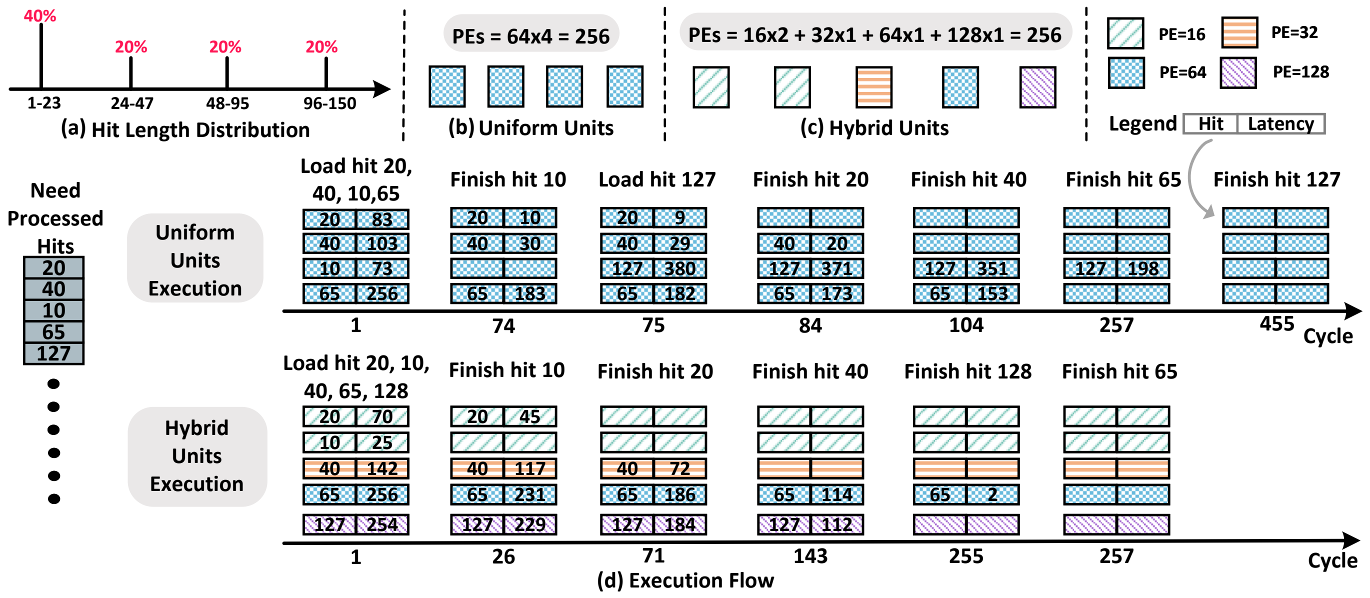

Fig. 9. A toy example of the hybrid unit strategy versus the unified unit strategy. (a) An assumed hit length distribution. (b) The uniform units strategy. (c) The hybrid units strategy. (d) The execution flow of (b) and (c).

for the banded Smith-Waterman algorithm [19]. Considering the diversity of hits, using different scales of bands for hits with different lengths can reduce the pressure of speculation-and-test and thus provide higher iso-area throughput.

D. Coordinator Design

Hits Fragmentation Problem. Since the subsequent scheduling request will read the next hits block with fixed *batch_size* and *offset*, the allocated failed hits[5] will be kept in *Hits Buffer* (cf. Fig. 10), which leads to a fragmentation problem. This problem will result in many hits not being allocated to EUs and cause the hits buffer to have less writable space.

Solution to Hits Fragmentation. We first define the allocated hits offset in the *Processing Buffer (PB)* (cf. Fig. 10) to know where to read the hits in the subsequent scheduling. In addition, to ensure the unprocessed hits of this scheduling can be processed later, we place the unprocessed hits at the end of the data batch and write them back to the original batch location in *PB*. With these two strategies, the unprocessed hits of this scheduling can be processed next time according to the offset and batch size.

Hits Allocation Problem. There are two basic resource allocation methods: (1) Allocating computing units in groups with the same number of PEs guarantees that the different groups do not interfere and that the optimal computing unit is always assigned to the hit. However, once the number of hits is more than idle resources, hits can not be allocated to resources, which affects the scheduling efficiency. (2) Allocating all computing units in one group ensures that all idle resources are shared, making it easier to allocate hits to idle computing units. Unfortunately, this approach is too aggressive and can easily lead to short hits being executed by large computing units, which will bring high execution latency.

Solution to Hits Allocation. We first group the EUs into several groups according to its PEs numbers. Similarly, we also divide the hits to be processed into several groups according to the hit length, and the grouping threshold depends on the grouping threshold of EUs. By using this strategy, adjacent resources can be supplemented to ensure scheduling efficiency when some specific resources are limited, and it also alleviates the occurrence of excessive computing latency.

Dataflow and Hardware Design. Fig. 10 shows the dataflow and hardware structure of the Coordinator. The Hits Buffer triggers a switch operation when the *Store Buffer (SB)* reaches a threshold (e.g., 75%). When the number of idle EUs reaches a threshold (e.g., 15%), an allocation process will be triggered to allocate hits in the *PB* to idle EUs. The allocation process is a low-latency greedy allocation process that allocates the optimal or sub-optimal EU for each hit. Unallocated hits are left for the next allocation process. It has the following nine steps. ❶ Load the unprocessed hits from the *PB* according to the current *offset* and constant *batch_size*. ❷ Compute the extension length of each hit, which is the difference between the end coordinate and the start coordinate of the *read_pos*. ❸ Sort the hits according to *hit_len*. ❹ Filter the table into two parts according to whether the *hit_len* is greater than a *split_threshold*. The *hit_len* (7, 29, 40) will be assigned to the upper group and *hit_Len* 103 to the other group. ❺ The computing units with 16 PEs and 32 PEs are grouped uniformly, and the computing units with 64 PEs and 128 PEs are grouped uniformly. ❻ Assign the hit to the optimal or near-optimal computing unit. ❼ Merge the table of

[5]The allocated failed hits mean that these hits are not allocated to executable EUs in the last scheduling. This frequently occurs since the length of hits can not be predicted in advance.

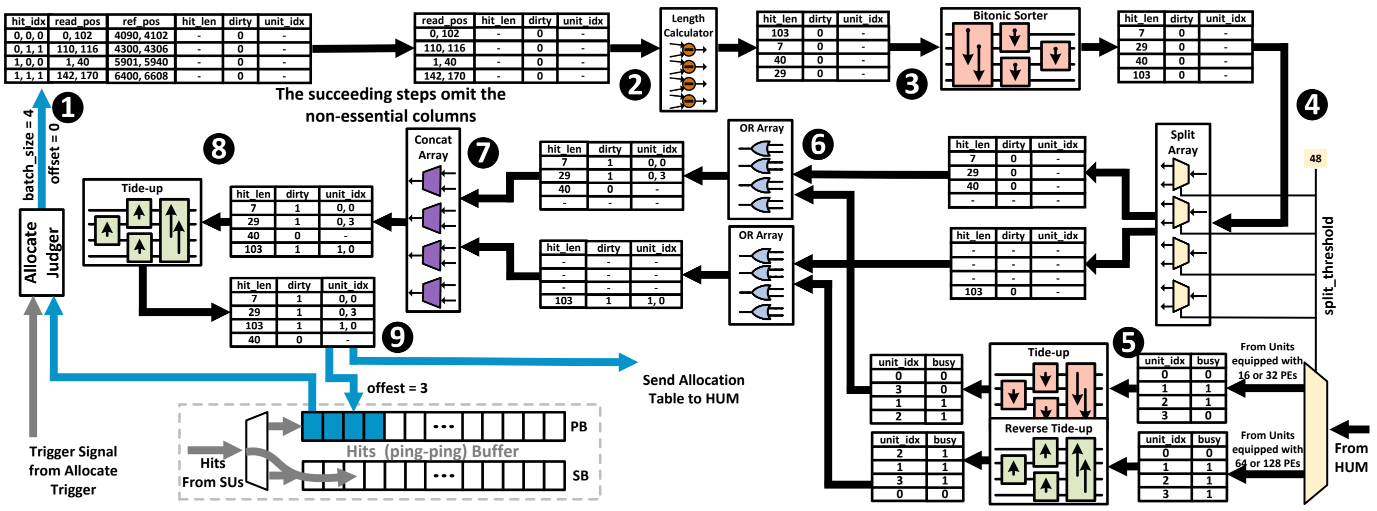

Fig. 10. The data flow and hardware design of the Coordinator.

the completed allocation, and the invalid rows will be filtered out. ❽ Move the allocated hits to the top and the unallocated hits to the bottom. ❾ The allocation result is written back to the *PB*, and the *offset* is adjusted to 3 to ensure that the un-allocated hit of *hit_len* 40 can be allocated next time.

V. Evaluation Results

A. Implementation of SUs and EUs

To measure the improvement of NvWa for throughput speedup and power consumption, we instantiate the SUs and EUs using two prior designs [60], [65] (noted as SUs+EUs in the following paper).

The SUs design is based on a bitwise and vectorized implementation of the FM-index search algorithm [65], and the FM-index interval is set to 128. The EUs design is modified from the open-source systolic array of Darwin [60]. The configuration of EUs is from the statistics of a most common dataset (i.e., NA12878 [2]), which guarantees a stable performance improvement even when the datasets are changed. As shown in Table I, we fix the number of SUs as 128 considering the previous work [60], [61], which can consume sufficient input data. We fix the number of PEs as 2880 since the amount of resources of EUs accounts for most of NvWa, and obtain the number of EUs for each class by solving Formula 5. As shown in Table I, the total number of EUs is 70, where the numbers of EUs for 16 PEs, 32P Es, 64 PEs and 128 PEs are 28, 20, 16, and 6, respectively.

Moreover, our algorithms and parameters of SUs and EUs are faithful to de facto standard software BWA-MEM (0.7.17-r1198-dirty) [14], [38], [43], [46], [53], [66], e.g., the scoring scheme, the affine gap penalty, and the trace-back support. This ensures NvWa has no loss of accuracy.

B. Methodology

We use the following strategies and tools to measure the performance and power consumption of NvWa.

Architecture Simulator. We build a cycle-accurate and execution-driven simulator using Python to model the micro-architectural behaviors and measure execution time in the number of cycles. We integrate our simulator with Ramulator [35] using SWIG [3] to simulate the behaviors of memory accesses.

CAD Tools. We implement each module of NvWa in Chisel3 [11] and generate Verilog. We synthesize each module using Synopsys Design Compiler with the SIMC 14 nm standard VT library to measure each module's area and critical path delay and estimate the power using Synopsys PrimeTime PX. The critical path delay is 0.9 ns, putting the NvWa comfortably at 1 GHz clock frequency.

Memory Measurements. We use Cacti 7.0 [12] to evaluate the area, power, and access latency of the on-chip scratchpad memory. Since Cacti 7.0 only supports down to 32 nm technology, we apply four different scaling factors as previously described in [52], [63] to convert them to 14 nm technology. The energy of HBM 1.0 is estimated at 7pJ/bit, as shown in [51], [68].

Benchmark Datasets. We evaluate NvWa using the major version of the human genome assembly, GRCh38 [4]. We use chromosomes 1-22, X, and Y by filtering the mitochondrial genomes, unmapped contigs, and unlocalized contigs. The dataset is the human genome NA12878 [2] (single-ended ERR194147_1.fastq), consisting of 787,265,109 reads of 101 base pairs (bp), and from which we randomly sampled 200,000 reads.

Baseline Platform. To compare the performance of NvWa with state-of-the-art studies, we evaluate the CPU baseline BWA-MEM [38] and the GPU baseline GASAL2 [5] on a Linux server equipped with two Intel(R) Xeon(R) E5-2620 v4 CPUs, 128 GB DDR4 memory, and an NVIDIA A100 GPU. Table I shows the system configurations for the above implementations.

We evaluate the performance of GenAx, GenCache, SeedEx, and ERT using data reported by the original work [23], [24], [49], [57]. By specifying the same benchmark dataset (NA12878 [2]), we can fairly compare the performance of NvWa with those studies.

TABLE I
SYSTEM CONFIGURATIONS OF CPUS, GPUS, AND NVWA.

	BWA-MEM	GASAL2	NvWa
Compute Unit	16 cores @ 2.10GHz	6912 cores @ 1.41GHz	128 SUs and 70 EUs @ 1 GHz
On-chip Memory	20MB	40MB	512 KB (SUs), 20 MB (EUs), and 150 KB (Coordinator)
Off-chip Memory	136.5GB/s DDR4	1555GB/s HBM v2.0	256GB/s HBM v1.0

C. Overall Results

Throughput. Fig. 11 shows the throughput of NvWa applied to the read alignment pipeline against various baselines. The throughput of NvWa is 49150 K reads/sec. NvWa achieves 493×, 200×, 151×, 12.11×, and 2.30× speedup against CPU-BWA-MEM [36], GPU-GASAL2 [5], FPGA-ERT+SeedEx [24], [57], ASIC-GenAx [23], and PIM-GenCache [49], respectively. For a fair comparison and considering the power consumption in the next paragraph, the throughput per Watt of NvWa is 52.62× of GenAx, and 13.50× of GenCache.

Compared to CPUs and GPUs, NvWa customizes SUs and EUs and provides much higher parallelism than CPUs and GPUs, and NvWa eliminates complex software and hardware stacks. Compared to FPGAs, NvWa provides a higher clock frequency, adopts efficient hardware schedulers, and avoids high hardware and software switching overhead. Compared to ASICs, only the design of computing components (i.e., SUs+EUs) is inferior to GenAx and GenCache, since they use modified hardware-friendly algorithms, such as the Hash-based search algorithm for the *seeding* phase. To this end, NvWa achieves low latency resource allocation through the One-Cycle Read Allocator and provides lower computing latency and higher parallelism for EUs through the Coordinator and the Hybrid Units Strategy. The Coordinator design effectively prevents system blocking and starvation. These strategies achieve better throughput than GenAx and GenCache.

Since the design of SUs+EUs only performs bit vectorization [65] for the *seeding* phase and systolic array [60] for the *seed-extension* phase, the performance is only 88.79% of GenAx and 16.93% of GenCache. By incorporating the NvWa optimization, the throughput improvement is 12.11× higher than GenAx and 2.30× higher than GenCache. In detail, the Hybrid Units Strategy, the One-Cycle Read Allocator, and the Hits Allocator provide 3.32×, 1.73×, and 2.38× speedup, respectively.

Power Consumption. Table II shows the detailed power consumption of NvWa. The total area of NvWa is 27.009 mm^2, and the total power consumption is 5.754 W. When

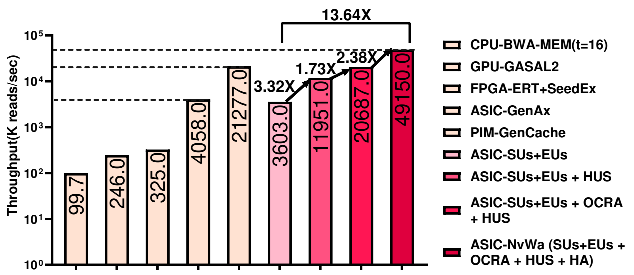

Fig. 11. Throughput comparison of NvWa to the state-of-the-art CPU [36], GPU [5], FPGA [24], [57], and ASICs [23], [49]. HUS denotes the Hybrid Units Strategy, OCRA denotes the One-Cycle Read Allocator, and HA denotes the Hits Allocator.

the HBM 1.0 is considered, the total power consumption is 7.685 W. The computing units (i.e., SUs and EUs) dominate the area and power consumption, accounting for 94.15% of the area and 86.61% of the power consumption. In contrast, the scheduling units have an area of only 1.58 mm^2 (5.84%) and a power consumption of only 0.77 W (13.38%). Compared to 16-threaded BWA-MEM, GASAL2, GenAx, and GenCache, NvWa achieves 14.21×, 5.60×, 4.34×, and 5.85× energy reductions, respectively[6]. Compared to CPU and GPU platforms with software and hardware stack switching and generic inefficient computing components, the custom hardware logic of NvWa is the main reason for its low power consumption. In contrast, the high throughput design used by GenAx and GenCache requires a large amount of SRAMs to cache data, which results in significant power consumption. NvWa employs resource-efficient computing units and improves system throughput with a lightweight scheduler, making NvWa the most energy-efficient accelerator so far.

TABLE II
AREA AND POWER BREAKDOWN OF INDIVIDUAL COMPONENTS IN NVWA.

Module	Category	Area(mm^2)	Power(W)
SUs[1]	Logic	0.5	0.36
	Table SRAM	2.16	0.71
EUs	Logic	1.62	0.30
	Table SRAM	21.15	3.614
Seeding Scheduler	SPM	0.13	0.04
	Logic	0.1	0.072
Extension Scheduler	Table SRAM	0.065	0.021
	Logic	0.23	0.165
Coordinator	SRAM Buffer	0.782	0.257
	Logic	0.273	0.215
Total	N/A	27.009	5.754

[1] Considering that SUs are bounded by memory, we use the LFMapBit architecture in [65] since it delivers sufficient throughput for our system.

[6]Since GenAx and GenCache do not consider the energy of memory, NvWa uses 5.693 W to compare power consumption with them.

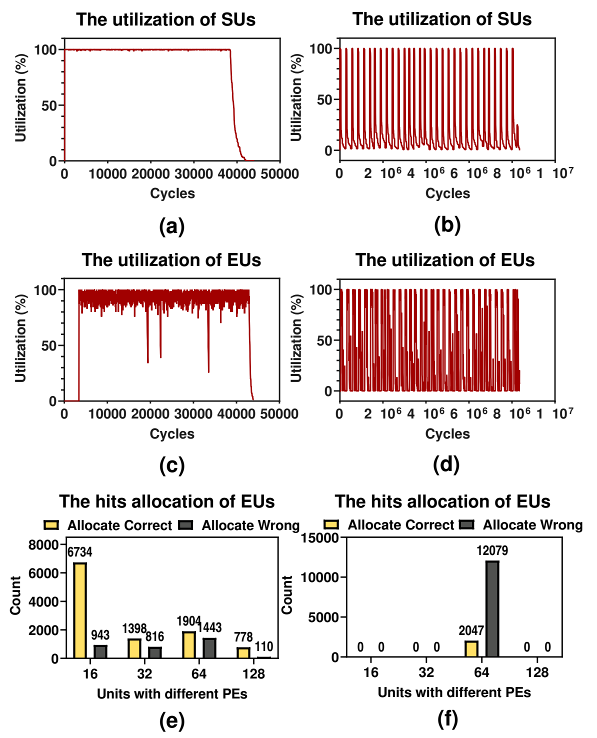

Fig. 12. Resource utilization improvements and comparisons. (a), (c), and (e) are the execution breakdown of NvWa. (b), (d), and (f) is the execution breakdown of SUs+EUs.

D. Resource Utilization and Optimization Analysis

Fig. 12 provides the resource utilization of key components related to the performance of NvWa, and the tested data here is 4000 reads of 101bp for better representation.

Fig. 12(a) and Fig. 12(b) show the resource utilization of SUs. The whole procedure is divided into loading time, running time, and emptying time. The loading time is only one cycle for our design, the average resource utilization of the running time is 97.1%, and the emptying time will account for a smaller proportion as the task size increases. The design of SUs+EUs employs the Read in Batch strategy, and such a coarse-grained approach fails to replenish data promptly for SUs that are idle after completing the current task, resulting in the utilization of only 23.51%.

As shown in Fig. 12(c), the loading time of EUs lags for some time since the *PB* sends bits to the EUs only after the buffer switch of stored bits is completed for the first time. The average resource utilization for the whole process is 85.36%. In addition, there are three spikes with utilization below 50% due to the random nature of the input data, and those last for a shorter period of cycles and do not significantly affect system performance. The design of SUs+EUs uses uniform EUs, which can not achieve optimal iso-area latency for different lengths of hits. As shown in Fig. 12(d), the utilization of EUs fluctuates terribly, with average utilization of only 32.31%.

Fig. 12(e) and Fig. 12(f) reflect whether each hit is correctly assigned to the optimal computing unit. 87.7% of the short but most numerous hits are correctly assigned to the computing unit of 16 PEs, and 87.6% of the long but less numerous hits are correctly assigned to the computing unit of 128 PEs. The percentage of units with 32 PEs and 64 PEs that are correctly allocated is lower, at 64.1% and 56.9%, respectively. Since the amount of this part is relatively small, which has little impact on the performance. Without using our strategy, only 14.5% of hits are correctly allocated, which causes low iso-area latency of EUs. More generally, based on the long execution latency problem reflected in Formula 3, using only a single scale of units (e.g., 16, 32, 64, 128) can not outperform our strategy.

E. Design Space Exploration

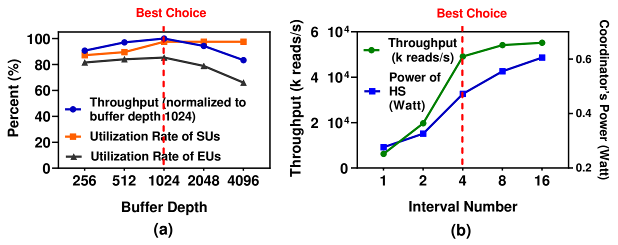

Fig. 13. Design space exploration. (a) The impact of different buffer depths on throughput, the utilization of SUs, and the utilization of EUs. (b) The impact of different intervals on throughput and the power consumption of the Coordinator.

Hits Buffer Depth. Fig. 13(a) demonstrates how the depth of the Hits Buffer affects the throughput, the average utilization of SUs, and the average utilization of EUs, respectively. When the buffer is small, the system will be more sensitive to the input data, and the system will be more susceptible to blocking or starving. When the buffer is in the blocking state, the SUs and EUs will move into the suspending state. And when the buffer is in the starving state, the EUs will enter the idle state. Both of these states eventually harm the system throughput. When the buffer is large, the time of the first buffer switch will be delayed, which is equivalent to postponing the time when EUs start running, and it affects the average utilization of EUs. The best result is achieved when the buffer depth is 1024, making a good trade-off between resource usage and system performance.

Interval Number. Fig. 13(b) shows the impact of the number of intervals on throughput and power consumption. A power of two is usually taken as the number of intervals for design simplicity. More intervals will generate more classes of hybrid EUs, and the number of each class is determined by solving Formula 5. For throughput, more intervals will provide better EUs for different lengths of hits, which can increase the throughput of the whole system. For the power consumption of the Coordinator, the buffer will dominate its power consumption when the interval is small, and the complex allocation logic will dominate its power consumption when the interval is large. We take an interval of four in our implementation as it provides the best trade-off between throughput and power consumption.

F. Sensitivity Analysis about Multiple Datasets

We used DWGSIM [31] to generate six reads datasets of NCBI reference genome [1] to evaluate the sensitivity of NvWa to other datasets. Fig. 14(a) shows the throughput speedup of NvWa on the short reads datasets versus the 16-threads CPU baseline. NvWa can achieve a speedup of 285.6× ∼ 357× on other datasets. This reflects the generality of our design. Fig. 14(b) illustrates the distribution of different short reads datasets under the four intervals. Since the error rates and algorithms for the second-generation sequencing are essentially stable, the different datasets have a roughly similar distribution to the NA12878 dataset. This is why NvWa can achieve stable performance on multiple datasets.

Fig. 14(a) also shows the throughput speedup on the long read datasets. Our design can still be applied to the long reads datasets by using the iterative scheme of GACT [60], [61]. Since the hit length distribution of long reads is different from short reads, the throughput speedup is 259× ∼ 272×. By adjusting the configuration according to the hit distribution of long reads, we argue that NvWa can still achieve considerable speedups on long reads datasets.

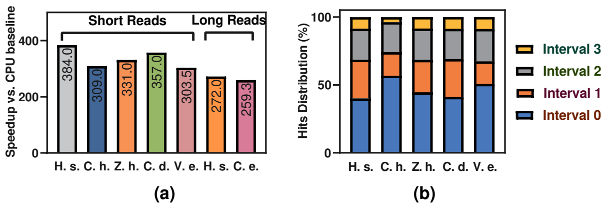

Fig. 14. (a) The performance of NvWa on multiple short and long reads datasets. H. s. denotes the Homo sapiens, C. h. denotes the Clitarchus hookeri, Z. h. denotes the Zapus hudsonius, C. d. denotes the Camelus dromedarius, V. e. denotes the Venustaconcha ellipsiformis, and C. e. denotes the Caenorhabditis elegans, respectively. The short reads datasets of H. s. still use the NA12878 dataset [2]. (b) The hits distribution percentage of multiple short reads datasets with four intervals.

VI. Discussion

Flexibility. The *seed-and-extend* paradigm is widely used in the 2nd/3rd-generation read alignment software [9], [30], [36], [39], [40], [48], [54]. The multifarious algorithms can benefit from NvWa if they follow the defined unified interface. As shown in Table III, the interface is composed of two parts. The data interface specifies the format standards for input and output to be followed by SUs and EUs. The control interface defines the states that the SU and EU need to support.

Long Reads. While our work focuses on short reads, the scalable design of NvWa also applies to long reads (≥ 1Kbp). Unlike the *seed-and-extend* paradigm used in short reads aligners [9], [38], [62], a handful of existing long reads aligners [39], [40] take the *seed-and-chain-then-fill* paradigm. It is expected that the *seed-and-chain-then-fill* paradigm will have the same execution diversity problem as shown in Sec. III since each input read has different characteristics.

TABLE III
THE UNIFIED INTERFACE DEFINITIONS OF NVWA

Interface Type	Unit Type	Direction	Signal Definition
Data	SUs	Input	[read_idx, read_metadata]
		Output	[read_idx, hit_idx, direction, read_pos, ref_pos] (noted as [sus_output])
	EUs	Input	[sus_output]
		Output	[sus_output, alignment_result]
Control	SUs	N/A	[idle, busy, stop]
	EUs	N/A	[idle, busy, stop, pe_number]

VII. Related Work

Accelerators for the Seeding or the Seed-Extension Phase. A large body of previous studies focused on accelerating the *seeding* phase using GPU [25], [59], FPGA [6], [10], [18], [21], [26], [64], ASIC [33], [57], [60], [61], [65], and Near-data processing (NDP) [32], [49], [69]. Two acceleration categories are usually employed to solve the random memory access problem of the *seeding* phase, e.g., using emerging device properties to accelerate the original algorithm [32], [69] and design a specialized algorithm-specific architecture [23], [33], [57], [60], [61]. A plethora of studies focused on accelerating the *seed-extension* phase using GPU [5], [25], FPGA [13], [20], [27], [29], [42], [44], [58], ASIC [23], [24], [47], [60], [61], and NDP [15], [34]. Similar to the *seeding* phase, these designs are divided into two categories, e.g., accelerating traditional dynamic programming algorithms [24], [60], [61] and accelerating novel matching algorithms [16], [23].

While these studies effectively address the memory-bound problem in the *seeding* phase and the compute-bound problem in the *seed-extension* phase, they did not fully consider the diversity problem. NvWa employs a scheduling approach that provides higher resource utilization for each unit with varying execution times.

Scheduling in the Sequence Alignment Accelerators. Previous work noticed the need for scheduling in the *seed-and-extend* paradigm. Darwin [60] and Darwin-WGA [61] used software API and data duplication to provide fine-grained control of the workloads on SUs and EUs. SeedEx [24] extended the multi-threading model in BWA-MEM to provide separate workers, which can run concurrently and communicate seed-extension accelerators over PCIe. ERT [57] utilized switch contexts to saturate memory bandwidth and employed the producer-consumer model to facilitate load-balancing in FPGA by adjusting the number of FPGA threads.

However, these studies can not substantially address the diversity problem mentioned in this paper. In our paper, we proposed the Seeding Scheduler to address the resource competition problem in the centralized buffer mechanism [23] and supply data for each idle SUs in only one cycle. And we proposed the Extension Scheduler and the Coordinator to control the hits and EUs at a fine-grained level, providing high parallelism and low latency for the *seed-extension* phase.

VIII. Conclusion

Sequence alignment is an essential step in genome data analysis. Many hardware efforts aimed to improve the execution latency in the workflow. To address the execution diversity problem and improve parallelism and resource utilization, we propose NvWa, which contains three novel scheduling mechanisms and corresponding architecture that target the *seeding* phase, the *seed-extension* phase, and the interaction, respectively. Moreover, NvWa has no loss of accuracy and is orthogonal to previous work. Experimental results show that NvWa can achieve 493×, 200×, 12.11×, 2.30× speedup and 14.21×, 5.60×, 4.34×, 5.85× energy reduction when compared with a 16-thread CPU baseline, an NVIDIA A100 GPU baseline, and two state-of-the-art accelerators, respectively.

IX. Acknowledgements

This work is supported by the National Key R&D Program of China under Grant No. 2022YFB4500403, the NSF of China under Grant No. T2125013 and 62202454, and the China National Postdoctoral Program for Innovative Talents under Grant No. BX2021320. We thank the anonymous reviewers who made suggestions for this paper. We are also inspired by Prof. Ninghui Sun's idea of loose coupling.

References

[1] "Genome," https://www.ncbi.nlm.nih.gov/data-hub/genome/.

[2] "NA12878 — IGSR sample," https://www.internationalgenome.org/data-portal/sample/NA12878.

[3] "Simplified Wrapper and Interface Generator," https://www.swig.org/.

[4] "Genome Reference Consortium Human Build 38 patch release 13 (GRCh38.p13) - Genome Assembly NCBI," https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39#, 2021.

[5] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "GASAL2: A GPU accelerated sequence alignment library for high-throughput NGS data," *BMC Bioinformatics*, vol. 20, no. 1, p. 520, Dec. 2019.

[6] N. Ahmed, V.-M. Sima, E. Houtgast, K. Bertels, and Z. Al-Ars, "Heterogeneous hardware/software acceleration of the BWA-MEM DNA alignment algorithm," in *ICCAD*, 2015, p. 7.

[7] M. Alser, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating Genome Analysis: A Primer on an Ongoing Journey," *IEEE Micro*, p. 12, 2020.

[8] M. Alser, J. Rotman, D. Deshpande, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer, B. Balliu, D. Koslicki, P. Skums, A. Zelikovsky, C. Alkan, O. Mutlu, and S. Mangul, "Technology dictates algorithms: Recent developments in read alignment," *Genome Biology*, vol. 22, no. 1, p. 249, Aug. 2021.

[9] S. F. AltschuP, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, p. 8, 1990.

[10] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging FPGAs for Accelerating Short Read Alignment," *IEEE/ACM Trans. Comput. Biol. and Bioinf.*, vol. 14, no. 3, pp. 668–677, May 2017.

[11] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC*. San Francisco, California: ACM Press, 2012, p. 1216.

[12] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 1–25, Jul. 2017.

[13] S. S. Banerjee, M. El-Hadedy, J. B. Lim, Z. T. Kalbarczyk, D. Chen, S. S. Lumetta, and R. K. Iyer, "ASAP: Accelerated Short-Read Alignment on Programmable Hardware," *IEEE Transactions on Computers*, vol. 68, no. 3, p. 16, 2019.

[14] S. Byma, S. Whitlock, L. Flueratoru, E. Tseng, C. Kozyrakis, E. Bugnion, and J. Larus, "Persona: A High-Performance Bioinformatics Framework," in *USENIX ATC*, 2017, pp. 153–165.

[15] D. S. Cali, "Accelerating Genome Sequence Analysis via Efficient Hardware/Algorithm Co-Design," Ph.D. dissertation, Carnegie Mellon University, Nov. 2021.

[16] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*. IEEE/ACM, 2020.

[17] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi, G. Singh, J. Gómez-Luna, N. A. Alserr, M. Alser, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "SeGraM: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 638–655.

[18] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu, "The SMEM Seeding Acceleration for DNA Sequence Alignment," in *FCCM*. Washington, DC, USA: IEEE, May 2016, pp. 32–39.

[19] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band," *Bioinformatics*, vol. 8, no. 5, pp. 481–487, 1992.

[20] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A Novel High-Throughput Acceleration Engine for Read Alignment," in *FCCM*. IEEE, May 2015, pp. 199–202.

[21] J. Cong, L. Guo, P.-T. Huang, P. Wei, and T. Yu, "SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing," in *FPL*. Dublin, Ireland: IEEE, Aug. 2018, pp. 210–2104.

[22] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *FOCS*. IEEE, 2000, pp. 390–398.

[23] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "GenAx: A Genome Sequencing Accelerator," in *ISCA*. Los Angeles, CA: ACM/IEEE, Jun. 2018, pp. 69–82.

[24] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, and R. Das, "SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space," in *MICRO*. Athens, Greece: IEEE/ACM, 2020, pp. 937–950.

[25] S. D. Goenka, Y. Turakhia, B. Paten, and M. Horowitz, "SegAlign: A Scalable GPU-Based Whole Genome Aligner," in *SC*, 2020, p. 13.

[26] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU," in *FCCM*. San Diego, CA, USA: IEEE, Apr. 2019, pp. 127–135.

[27] X. Guo, H. Wang, and V. Devabhaktuni, "A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm," *ISRN Bioinformatics*, vol. 2012, pp. 1–11, 2012.

[28] T. J. Ham, D. Bruns-Smith, B. Sweeney, Y. Lee, S. H. Seo, U. G. Song, Y. H. Oh, K. Asanovic, J. W. Lee, and L. W. Wills, "Genesis: A Hardware Acceleration Framework for Genomic Data Analysis," in *ISCA*. Los Angeles, CA: ACM/IEEE, 2020.

[29] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "A Banded Smith-Waterman FPGA Accelerator for Mercury BLASTP," in *FPL*, Aug. 2007, pp. 765–769.

[30] R. S. Harris, "Improved Pairwise Alignmnet of Genomic DNA," Ph.D. dissertation, The Pennsylvania State University, 2007.

[31] N. Homer, "Nh13/DWGSIM," Mar. 2021.

[32] 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*. Columbus OH USA: IEEE/ACM, Oct. 2019, pp. 587–599.

[33] L. Jiang and F. Zokaee, "EXMA: A Genomics Accelerator for Exact-Matching," in *HPCA*. IEEE, 2021.

[34] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A Resistive CAM Processing-in-Storage Architecture for DNA Sequence Alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.

[35] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, Jan. 2016.

[36] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.

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

[38] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv:1303.3997 [q-bio]*, May 2013.

[39] H. Li, "Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, Jul. 2016.

[40] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences," *Bioinformatics*, p. 7, 2018.

[41] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, Mar. 2010.

[42] I. T. Li, W. Shum, and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinformatics*, vol. 8, no. 1, p. 185, Jun. 2007.

[43] X. Li, G. Tan, B. Wang, and N. Sun, "High-performance genomic analysis framework with in-memory computing," in *PPoPP*. Vienna, Austria: ACM Press, 2018, pp. 317–328.

[44] Y.-L. Liao, Y.-C. Li, N.-C. Chen, and Y.-C. Lu, "Adaptively Banded Smith-Waterman Algorithm for Long Reads and Its Hardware Accelerator," in *ASAP*. Milan: IEEE, Jul. 2018, pp. 1–9.

[45] R. J. Lipton and D. P. Lopresti, *Comparing Long Strings on a Short Systolic Array*. Princeton University, Department of Computer Science, 1986.

[46] M. Lo, Z. Fang, J. Wang, P. Zhou, M.-C. F. Chang, and J. Cong, "Algorithm-Hardware Co-design for BQSR Acceleration in Genome Analysis ToolKit," in *FCCM*. Fayetteville, AR, USA: IEEE, May 2020, pp. 157–166.

[47] A. Madhavan, T. Sherwood, and D. Strukov, "Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms," in *ISCA*. ACM/IEEE, 2014, p. 12.

[48] J. Marić, I. Sović, K. Križanović, N. Nagarajan, and M. Šikić, "Graphmap2 - splice-aware RNA-seq mapper for long reads," Bioinformatics, Preprint, Jul. 2019.

[49] A. Nag, C. N. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment," in *MICRO*. Columbus OH USA: ACM, Oct. 2019, pp. 334–346.

[50] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[51] M. O'Connor, "Highlights of the high-bandwidth memory (hbm) standard," 2014.

[52] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy Efficient Architecture for Graph Analytics Accelerators," in *ISCA*, 2016, p. 12.

[53] A. Prabhakaran, B. Shifaw, M. Naik, and P. Narvaez, "Infrastructure for Deploying GATK Best Practices Pipeline," Intel, Tech. Rep., 2015.

[54] 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, Dec. 2004.

[55] M. Schmidt, K. Heese, and A. Kutzner, "Accurate high throughput alignment via line sweep-based seed processing," *Nat Commun*, vol. 10, no. 1, p. 1939, Dec. 2019.

[56] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981.

[57] A. Subramaniyan, J. Wadden, K. Goliya, N. Ozog, X. Wu, S. Narayanasamy, D. Blaauw, and R. Das, "Accelerated Seeding for Genome Sequence Alignment with Enumerated Radix Trees," in *ISCA*. ACM/IEEE, 2021.

[58] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, "Accelerating Millions of Short Reads Mapping on a Heterogeneous Architecture with FPGA Accelerator," in *FCCM*, 2012, p. 4.

[59] J. S. Torres, I. B. Espert, and I. M. Castello, "Using GPUs for the Exact Alignment of Short-Read Genetic Sequences by Means of the Burrows-Wheeler Transform," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 4, p. 12, 2012.

[60] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly," in *ASPLOS*. Williamsburg, VA, USA: ACM, 2018, pp. 199–213.

[61] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-WGA: A Co-processor Provides Increased Sensitivity in Whole Genome Alignments with High Speedup," in *HPCA*. Washington, DC, USA: IEEE, Feb. 2019, pp. 359–372.

[62] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 314–324.

[63] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the Power Wall: A Path to Exascale," in *SC*. New Orleans, LA, USA: IEEE, Nov. 2014, pp. 830–841.

[64] H. M. Waidyasooriya and M. Hariyama, "Hardware-Acceleration of Short-Read Alignment Based on the Burrows-Wheeler Transform," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1358–1372, May 2016.

[65] Y. Wang, X. Li, D. Zang, G. Tan, and N. Sun, "Accelerating FM-index Search for Genomic Data Processing," in *ICPP*. Eugene, OR, USA: ACM, 2018, pp. 1–12.

[66] L. Wu, D. Bruns-Smith, F. A. Nothaft, Q. Huang, S. Karandikar, J. Le, A. Lin, H. Mao, B. Sweeney, K. Asanovic, D. A. Patterson, and A. D. Joseph, "FPGA Accelerated INDEL Realignment in the Cloud," in *HPCA*. Washington, DC, USA: IEEE, Feb. 2019, pp. 277–290.

[67] Y.-C. Wu, Y.-L. Chen, C.-H. Yang, C.-H. Lee, C.-Y. Yu, N.-S. Chang, L.-C. Chen, J.-R. Chang, C.-P. Lin, H.-L. Chen, C.-S. Chen, J.-H. Hung, and C.-H. Yang, "21.1 A Fully Integrated Genetic Variant Discovery SoC for Next-Generation Sequencing," in *ISSCC*, 2020, pp. 322–324.

[68] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *HPCA*. IEEE, 2020, p. 15.

[69] F. Zokaee, M. Zhang, and L. Jiang, "FindeR: Accelerating FM-Index-Based Exact Pattern Matching in Genomic Sequences through ReRAM Technology," in *PACT*, Sep. 2019, pp. 284–295.