SALoBa

Maximizing Data Locality and Workload Balance for Fast Sequence Alignment on GPUs

Park, Seongyeon ; Kim, Hajin ; Ahmad, Tanveer; Ahmed, Nauman ; Al-Ars, Zaid ; Hofstee, Peter; Kim, Youngsok ; Lee, Jinho

DOI
10.1109/IPDPS53621.2022.00076

Publication date
2022

Document Version
Final published version

Published in
Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

SALoBa: Maximizing Data Locality and Workload Balance for Fast Sequence Alignment on GPUs

Seongyeon Park[†], Hajin Kim[†], Tanveer Ahmad[§], Nauman Ahmed[§],
Zaid Al-Ars[§], H. Peter Hofstee[§‡], Youngsok Kim[†], and Jinho Lee[†*]
[†]*Yonsei University* [§]*TU Delft* [‡]*IBM*
{syeonp, kimhajin},@yonsei.ac.kr, {t.ahmad, n.ahmed, z.al-ars}@tudelft.nl,
hofstee@us.ibm.com, {youngsok, leejinho}@yonsei.ac.kr

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00076

***Abstract*—Sequence alignment forms an important backbone in many sequencing applications. A commonly used strategy for sequence alignment is an approximate string matching with a two-dimensional dynamic programming approach. Although some prior work has been conducted on GPU acceleration of a sequence alignment, we identify several shortcomings that limit exploiting the full computational capability of modern GPUs. This paper presents SALoBa, a GPU-accelerated sequence alignment library focused on seed extension. Based on the analysis of previous work with real-world sequencing data, we propose techniques to exploit the data locality and improve workload balancing. The experimental results reveal that SALoBa significantly improves the seed extension kernel compared to state-of-the-art GPU-based methods.**

***Index Terms*—Genome sequencing, Sequence alignment, Smith-Waterman, GPU acceleration**

I. Introduction

With the fast advances in next-generation sequencing (NGS) techniques, the monetary cost for DNA sequencing has been declining at a rate that is outpacing Moore's law [1]. However, on the other side of the coin, this rapid throughput in sequencing means that data processing has become a more severe bottleneck. For example, performing read mapping of the human genome on an Intel Xeon processor now takes more than 20× the sequencing time [24].

Read mapping, a process in sequence alignment, maps a piece of query DNA (generated from sequencing) to matching locations of a reference DNA. However, the reference and query DNAs do not exactly match due to sequencing errors and/or mutations. Therefore, the problem falls into the category of approximate string matching.

One popular strategy to solve this problem is *seed-and-extend*, where the seeding phase locates a few exact matches, and the extension phase performs approximate matching based on dynamic programming (DP) such as the Smith–Waterman algorithm [57]. Unfortunately, the Smith–Waterman algorithm has quadratic complexity with the input length, which continues to increase as NGS techniques evolve.

In this work, we focus on the extension, which consumes a significant portion of the execution time for read mapping [12]. Many proposals have been made for dedicated hardware accelerators with ASICs [22], [23], [58] or FPGAs [10], [25], [26], [28], [62] to cope with the computational bottleneck problem. However, such accelerators are yet to be widely used in practice. The ASICs are expensive to produce and would have difficulty appearing in the market unless mass production was guaranteed. The FPGAs are less prone to this issue because many FPGA accelerator cards are already available on the market. However, FPGA accelerator cards are not yet the dominant option because they are expensive and hard to program.

Under such circumstances, GPU-based accelerations can be viable options because they are cheaper, easier to find on the market, and require less effort to develop software. Therefore, many GPU-based libraries have been developed for bioinformatics [3], [9], [48]. With the fast growth in the computational capabilities of GPUs [17], GPUs will likely continue to be the primary option for accelerating sequence alignments.

However, by analyzing the state-of-the-art GPU-accelerated seed extension, we identified several missing opportunities for performance improvement that have become critical, especially with long string queries. Considering that the lengths of sequence reads are rapidly growing with third-generation sequencers [55], the performance gap between the ideal case and the existing software will widen. Specifically, we found that existing GPU kernels 1) inefficiently utilize memory and 2) suffer from load imbalance.

In this paper, we present SALoBa (**S**equence **A**lignment with Data **Lo**cality and Workload **Ba**lance), which addresses the mentioned problems to achieve superior performance. Taking lessons from several ASIC/FPGA accelerators [10], [23], [62], SALoBa puts together a kernel that makes better use of the computational capability of modern GPUs.

SALoBa utilizes intra-query parallelism by allocating multiple CUDA threads to a query-reference pair. Even though this option has been studied by some prior art [13], [35], these suffer from inefficient memory access patterns and resource underutilization, being outperformed by the current state-of-the-art, which only uses inter-query parallelism. SALoBa addresses those issues by introducing two novel techniques: lazy spilling to global memory and subwarp scheduling. In *lazy spilling*, the data are first accumulated to the CUDA shared memory and later spilled to the global memory in a coalesced manner. By utilizing the double-buffered shared memory region in a rotating manner, the redundancy in global

*Corresponding author

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00076

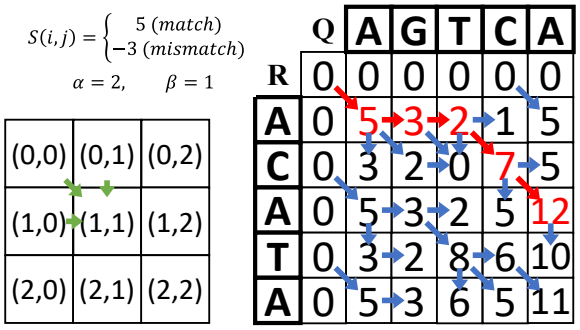


Fig. 1: An example seed extension algorithm.

memory is greatly reduced without much overhead to the shared memory. In addition, *subwarp scheduling* divides a warp into multiple subwarps to mitigate the underutilization problem. While this slightly increases the workload imbalance, the gain from better resource utilization often dominates the overhead from workload imbalance.

According to the experimental results, SALoBa performs significantly faster than the state-of-the-art GPU aligner in the seed extension kernel over several sequence lengths. When tested on real-world datasets with various lengths, we obtained superior results due to better workload balancing. Our contributions can be summarized as follows:

- We identify several unexploited opportunities from the current libraries for performance improvements.
- We propose SALoBa, which provides significant speedup compared to the current state-of-the-art GPU-based alignment libraries.
- We perform an extensive amount of evaluation, including synthetic and real-world data from a popular read alignment software to demonstrate the efficiency of SALoBa.

II. Background

A. Seed Extension

The heart of the sequence alignment problem is approximate string matching. Because there could be multiple possible paths to take every time a mismatch occurs, its complexity quickly increases along with the length of the input pair.

The seed-and-extend strategy is an approach to perform alignment efficiently. Based on the observation that a good alignment usually contains many matches, the strategy first finds multiple exact matches, called *seeds*. Based on these, approximate matching scores are calculated by *extending* to both directions from the found seeds.

Seed extension is often performed using a DP algorithm that fills a two-dimensional DP table with the complexity of $O(N^2)$. Popular algorithms are Smith–Waterman [57] and Needleman–Wunsch [51]. The score of each cell is calculated based on predetermined scores for the type of differences (i.e., insert, delete, and mismatch). With some adjustments for considerations for frequent long gaps, the affine gap function calculates the score of a DP cell $H(i,j)$ as follows:

$$H(i,j) = max \begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ H(i-1,j-1) + S(i,j) \end{cases}, \quad (1)$$

$$E(i,j) = max \begin{cases} H(i,j-1) - \alpha \\ E(i,j-1) - \beta \end{cases}, \quad (2)$$

$$F(i,j) = max \begin{cases} H(i-1,j) - \alpha \\ F(i-1,j) - \beta \end{cases}, \quad (3)$$

where $S(i,j)$ is a score function that returns a positive value when the reference at i and query at j match and returns a negative value otherwise. In addition, E and F are auxiliary variables that keep track of the continuing gaps. Last, α and β account for different gap penalties according to new gaps or continued gaps, respectively.

Fig. 1 presents an example DP table, where Q represents a query string, and R represents a reference string. To calculate a cell, the cells from three adjacent cells, the top, left, and top left, are needed, represented by green arrows. The trace of the highest score in the DP table represents the best match found by the algorithm, denoted with red arrows and numbers.

B. Baseline GPU-based Seed Extension

There have been several attempts to accelerate seed extension using GPUs. In sequence read data, each base is represented by a character data type of eight bits. However, only five bases exist within the sequences: A, C, G, T/U, and N, where T is used for DNA and U is used for RNA. The N denotes unknown bases. Having five bases indicates that at least three bits are needed to represent each. Because three-bit representations are inefficient to deal with in modern architectures, a four-bit packed representation is often used [3], [9] and some work utilizes eight-bit representation [35].

Existing methods for GPU-based seed extension can be roughly categorized by the parallelism they utilize: *intra-query parallelism* and *inter-query parallelism*. Intra-query parallelism refers to processing multiple cells in a DP table concurrently. As shown in Fig. 1, parallelism exists in an anti-diagonal form in the DP table. For example, after the cell (0,0) has been processed, cells (0,1) and (1,0) can be processed in parallel. Afterwards, cells (0,2), (1,1), and (2,0) can be processed in parallel. Many ASIC/FPGA accelerators successfully take advantage of this, commonly using one-dimensional systolic array structures [10], [23], [62]. Some GPU-based approaches utilize this type of parallelism [13], [35]. However, they often fail to achieve sufficient speedup due to resource under-utilization and inefficient memory access [13]. On the other hand, inter-query parallelism refers to simply processing multiple query-reference pairs in parallel. Because the latter is easier to optimize resource utilization, many successful libraries adopt this approach [3], [9], [39], [45].

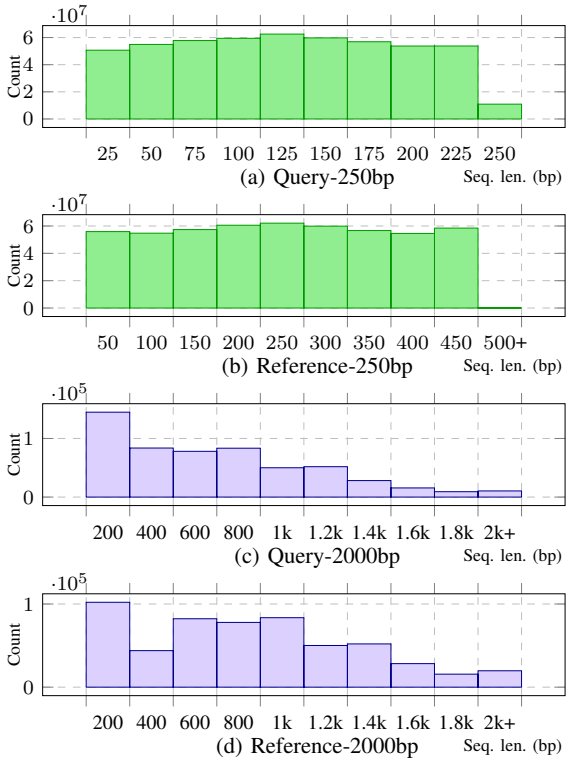


Fig. 2: Distribution of the sequences as input to the seed extension in BWA-MEM [37].

Among the libraries using inter-query parallelism, GASAL2 [9] is known to show the current state-of-the-art performance. In this section, we describe the details of its strategy. The GPU registers are 32-bit wide; thus, eight bases from each query and reference are fetched in a single step. Therefore, it is natural to process 8×8 cells at once that correspond to a single-word reference and a single-word query. After processing the block of 8×8 cells, the thread advances to the right. For this, a column of eight rightmost cells must be stored for dependency. The dependency is fulfilled by keeping the values of these DP cells within the registers. When a thread reaches the rightmost end of the table, it moves to the bottom part of the DP table. Then, the cells from the top must be accessed for dependency, and the thread stores the bottom eight cells of each 8×8 block in the global memory. Thus, considering the matrix size of $N \times N$, the amount of global memory access becomes $N \times N/4$ for reading and writing dependent cell values.

III. Motivation - Diagnosis

In this section, we diagnose the state-of-the-art aligner GASAL2 [9] and discuss potential opportunities for further performance improvement. First, we reveal a significant load imbalance between individual queries. Second, we identify that the use of global memory is not coalesced, which requires a significant amount of redundant memory access.

TABLE I: Amount of Data Stored and Accessed for the Existing GPU Aligner

Data	Quantity
Necessary	$2N$
Stored	$2N + N^2/4$
Accessed (Until Pascal [4])	$128N + 16N^2$
Accessed (After Volta [5])	$32N + 4N^2$

A. Load Imbalance

As discussed in Section II-B, the main parallelization scheme of GASAL2 is inter-query parallelism. In the strategy of letting each CUDA thread handle a single query, the main advantage is that there is no complicated inter-thread synchronization or costly communication.

However, one drawback of this approach is that it ignores the variance in the workload for the seed extension. The seeding step provides the query and reference sizes as input to the seed extension. Because of this, the lengths vary by individual inputs, and a significant imbalance occurs.

Fig. 2 plots the distribution of the query and reference sequences from two types of reads (see Section V for detailed settings) from the popular alignment software BWA-MEM [37]. As illustrated, both distributions range from zero to several hundred or thousand and are not well clustered, implying a substantial amount of warp divergence within GPUs. As depicted in the figure, the difference of length between the shortest and longest strings can be up to $10\times$ for both the query and reference string. Provided that the computational complexity is proportional to the multiple of the two string lengths, the workload imbalance could be very large in practice.

B. Memory Inefficiency

With four-bit sequence packing, it is rational to process 8×8 DP cells at a time. When moving to the next cell,[1] the current cell content can be captured within the registers. However, the dependency structure of the seed extension also requires long-term storage of the cell information to process the next row of 8×8 cells. This intermediate data well exceed the size of the register file and are usually stored in the global memory (i.e., DRAM).

However, this scheme incurs two inefficiencies. First, the intermediate data need not remain in the global memory at the end of the kernel and are considered overhead. Therefore, reducing this overhead contributes to better performance. Second, the minimum access size of the GPU's global memory is 128 B (or 32 B from Volta [5] architecture, as identified by [32]), whereas the individual cell data size is only 4 B. If not captured by the L2 cache, this will incur a number of redundant access instances for the same intermediate data, leading to an inefficient kernel.

[1]Without loss of generality, we assume that cells are processed left to right and top to bottom.

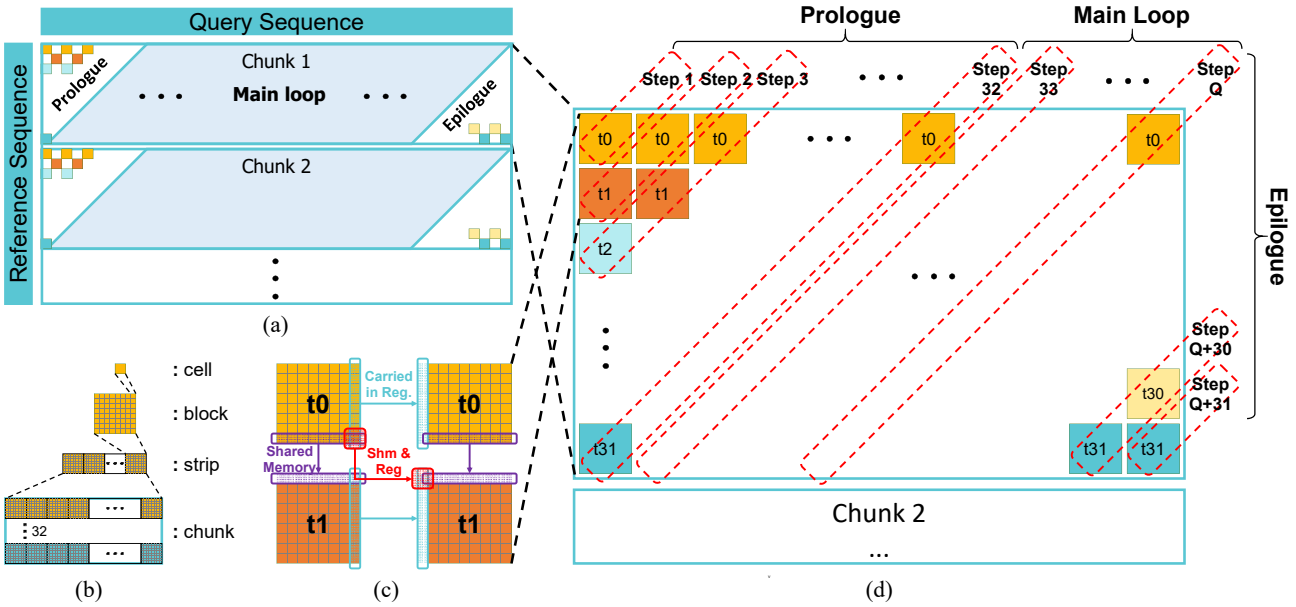


Fig. 3: Intra-query parallelism. (a) shows parallelization strategy using a warp, (b) illustrates the terms used in this figure. (c) demonstrates how the dependency between cells are handled and (d) depicts the use of anti-diagonal structure.

TABLE I lists the data volume, input data necessary for the extension (Necessary), data stored for intermediate data (Stored), and data accessed due to the access granularity (Accessed). The table reveals that the inefficiency is multifold even with modern architectures, which provides another opportunity for improvements. In this work, we demonstrate that SALoBa can remove much of this access, leading to superior performance.

IV. SALoBa Design

A. Intra-query Parallelism

Because each cell in the DP table has dependencies from the top, left, and top-left cells, it is known that intra-query parallelism exists in an anti-diagonal form, as depicted in Fig. 3. Adopting the idea, we use multiple CUDA threads to concurrently handle anti-diagonal elements. A slight difference is that instead of individual threads computing a single cell at a time, the cells are assigned to threads in 8×8 *blocks* because of the 4-bit packing on the reference and query sequences.

In the first version of the library, we decided that 32 threads at most—a warp—should collaborate in processing a query. Because intra-warp synchronization is relatively cheap or free (for GPUs before introduction of independent thread scheduling [5]), this becomes an attractive design choice. While using more than 32 threads is theoretically possible, it would require threadblock level synchronizations (i.e., `__syncthreads()`) that cause non-negligible overhead.

For communication between the threads, we used the CUDA shared memory. Using shared memory as a communication channel and reusing it every iteration, only a fixed amount of shared memory is needed, and all access to the shared memory is conflict-free (see Section IV-B).

Fig. 3 (a) and (b) illustrate the procedure of SALoBa. The largest units of computation are called *chunks*, which are horizontal partitions of the DP table. The chunks have the height of 32 blocks and the width of the entire query sequence.

As displayed in the figure, a thread processes a block in a *strip* (i.e., a row of blocks) of the chunk per step. In the first step, the first thread starts the processing by fetching a 32-bit word from each of the query and reference sequences, which is enough to process an 8×8 *block* of cells.

The number of ready-to-process cells increases by one per step in the upcoming steps. Therefore, the number of threads participating in the computation increases until the 32nd step, forming a 31-step long *prologue*. In the main loop stage,[2] all threads in the warp simultaneously process blocks in an anti-diagonal manner, advancing one block to the right per step. This process continues for $Q-31$ steps, where Q is the number of the blocks in the query sequence (i.e., $\lceil query\_length/8 \rceil$). When the first thread reaches the end of the row, the *epilogue* starts, and the number of participating threads decreases by one per step until the last thread reaches the end of the row. At the end of the epilogue, the total number of steps taken for processing the entire chunk is $Q+31$.

Each time the threads advance to the next step, they access the dependency data from the cells on the top, left, and top-left (Eq. 3). Fig. 3 (c) demonstrates how this is done. When a block is calculated, the data from the eight cells to the left are what the current thread computed in the previous step. Therefore,

[2]This pattern is also called the 'kernel,' but we decided to call it the main loop stage because it would be confusing with the term GPU kernel.

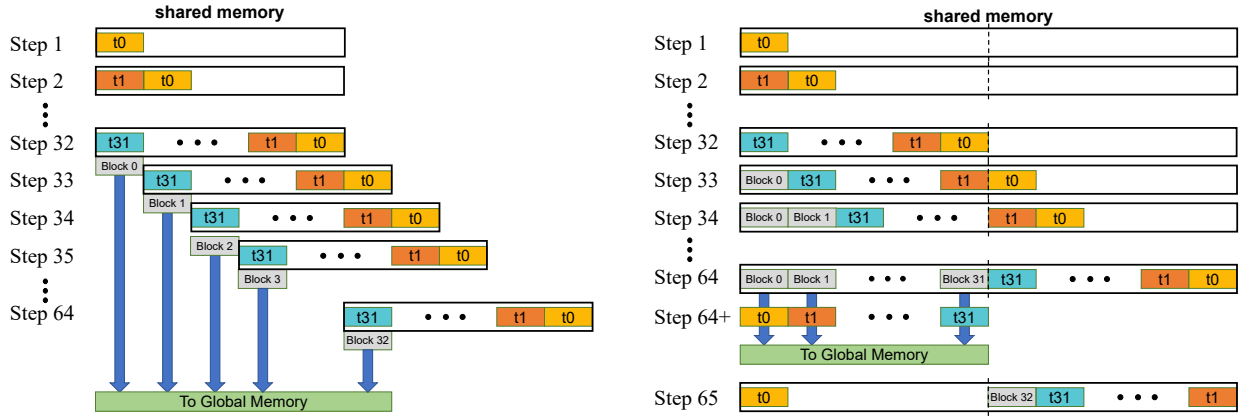


Fig. 4: Lazy spill optimization. (Left): Naive global memory access. (Right): Proposed optimized access. Each small rectangle represents 8×1 cells stored for dependency. A colored rectangle labeled 't#' denotes that it is being accessed by thread# and a gray rectangle labeled 'block #' represent that it is from block id # and is ready to be spilled to the global memory.

the data can be stored in the register. The data from the eight cells at the top were computed by the adjacent thread in the previous step. At the end of processing a block, each thread writes the bottom 8×1 cells in the shared memory. In the next step, each thread reads the data in the next position of the shared memory so that it can receive the 8×1 cells from the above block. Last, one cell at the top left has dependency on the current block, which is what the thread received from the shared memory in the previous step. Fortunately, it can also be passed using the register. Thus, the number of cells stored in the register becomes nine instead of eight.

This scheme not only reduces the warp divergence but also improves memory access. In the previous technique, all bottom cells in each strip must be stored and read back to/from the global memory. In contrast, with intra-query parallelism, only the bottom cells of each chunk (not of the strip) are stored and read to/from the global memory. For a 32-thread warp, this reduces the amount of intermediate data access to 1/32.

B. Lazy Spill to Global Memory Access

For processing a chunk, the data from the bottom-most cells must be stored for processing the top-most cells of the next chunk. Usually, this easily exceeds the shared memory capacity and must be stored in the global memory. In a naive scheme, as in Fig. 4 (left), the last thread ($t31$) stores the bottom-most cells of a block in the global memory, and the first thread ($t0$) reads them from the global memory as the next chunk is processed. However, as diagnosed in Section III, this results in an abundant amount of non-coalesced access, becoming a reason for inefficiency.

To address this problem, we used *lazy spilling* implemented using double-buffered shared memory for each warp to reduce the amount of global memory access. Fig. 4 (right) indicates the shared memory location where the threads write at each step. For each warp, a shared memory region is allocated with the size of $2 \cdot dim(block) \cdot \#threads$. As explained in Section IV-A, the first thread ($t0$) starts writing on the first location of the shared memory, which is passed to the second thread before the next step. In each step, the threads read data from the shared memory, process a block, and overwrite the bottom cells in the block to the shared memory. Then, all threads shift one block location for the next step.

After the prologue, the threads leave a data trail, written by the last ($t31$) thread. Luckily, these are the data to be stored on the global memory. When the first thread ($t0$) reaches the end of the buffer at step 64, the shared memory has a consecutive region of data at the left side which are large enough to be processed by a warp. The threads stop processing and gather together to spill the data to the global memory in a coalesced write. Afterward, the threads wrap around in the shared memory, and the trail of block data starts forming on the right half of the shared memory, which is later spilled to the global memory.

The spilled data from the previous chunk must be read to process the next chunk so that the first thread can use the data to calculate the cell values. This process is also performed in the same way using the double-buffered shared memory in the opposite direction of the writes.

C. Subwarp Scheduling: Trade-off Between Parallelism and Workload Balancing

The proposed strategy—a warp per query—can eliminate the workload imbalance between threads in a warp because they work together. However, as demonstrated in Section IV-A, a problem exists in the prologue and epilogue. The average thread utilization over these phases is 50% (because it has a triangular form), and their size is proportional to the warp size. The DP table must be significantly larger than the number of threads in a warp to amortize this overhead. However, according to Fig. 2, the size of the table dimension is only around several hundred, which is only a few times larger than the size of 32, and sometimes even smaller.

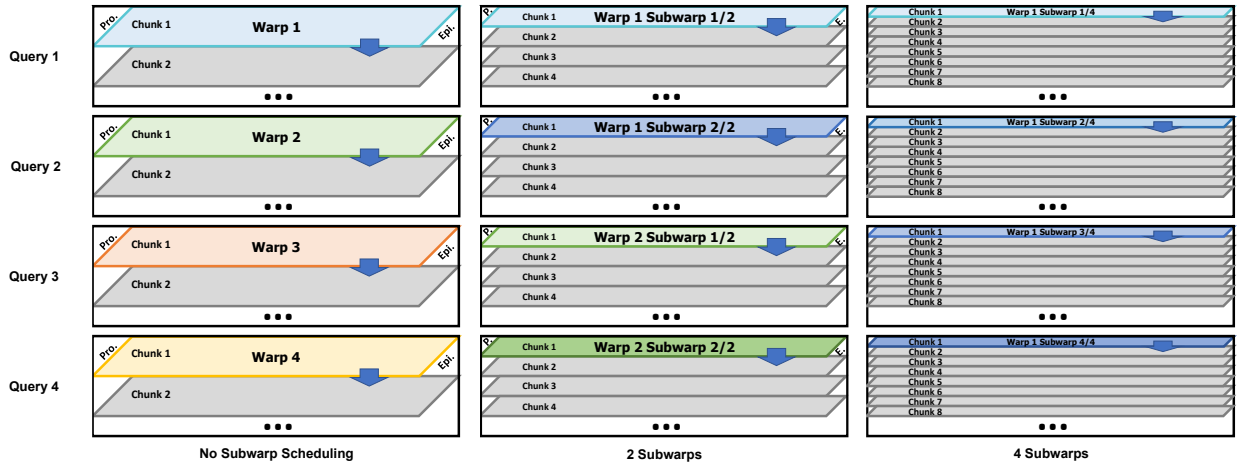


Fig. 5: Subwarp scheduling.

We chose to split the warp into multiple *subwarps* and let them process a single query, as in Fig. 5, similar to the idea of VWC [27] used in graph processing. Its effect is essentially a trade-off between workload balancing and resource utilization.

When many subwarps are used (i.e., the subwarp size is smaller), the sizes of the prologue and epilogue become smaller; therefore, the average utilization increases. However, multiple subwarps make the scheme suffer more often from warp divergence due to workload imbalance. In contrast, when fewer subwarps are used (i.e., the subwarp size is larger), it is less likely to suffer from imbalance at the cost of lower utilization from increased portions of the prologue and epilogue. Empirically, we found that using two or four subwarps yields the best results in our setting (see Section V).

One potential drawback is that using subwarps complicates the memory access optimization. If the same double-buffered technique is used, the number of threads accessing the consecutive memory address becomes less than 32. Subwarps larger than eight threads do not incur a considerable problem for architectures later than NVIDIA Volta [32]. For older architectures, the problem can be solved by allocating $N+32$ slots of shared memory instead of $2N$. By making the active region rotate around and making all threads in the entire warp write to 32 slots of data to the global memory together, a full coalescing can be achieved at the expense of a slightly higher shared memory capacity requirement.

V. Evaluation

A. Experimental Setup

We evaluated SALoBa on two platforms. First, as an 'affordable' system, we used a GTX1650 GPU card based on the Turing [6] architecture. The server has a six-core Intel i5-9600K CPU with 16 GB of RAM. We also conducted experiments on a 'high-end' system with an RTX3090 GPU card based on the Ampere [17] architecture. The server has a single socket, 12-core AMD EPYC 7272 CPU and 128 GB of RAM. Both machines run on Ubuntu 18.04 with CUDA version 11.2 and NVIDIA driver 460.27.04.

For comparison, we used the seed extension kernels from the following libraries as the baseline listed in TABLE II. **SOAP3-dp [50]** is an early algorithm for GPU-assisted short read alignment, which utilizes inter-query parallelism. **CUSHAW** is a family of GPU-assisted short read alignment algorithms. While most of its members [43], [44], [48] use GPU only for seeding phase, **CUSHAW2-GPU** [45] accelerates the seed extension part with a GPU using a similar strategy to SOAP3-dp. Better performance is obtained by compacting the global memory storage format and using CUDA texture memory. **NVBIO [3]** is a library of reusable components designed by NVIDIA to accelerate bioinformatics applications using CUDA. **SW# [35]** is an algorithm that utilizes intra-query parallelism. It splits the DP table into multiple anti-diagonal partitions and processes one partition per each kernel launch. Because it launches the kernel multiple times per query, it targets very long sequences. **ADEPT [13]** is the most recent work among algorithms that use intra-query parallelism. It uses shuffle instruction along with binary masking to make use of the anti-diagonal parallelism. Lastly, **GASAL2 [9]** is the state-of-the-art library. It achieves superior performance by further executing the sequence packing on GPU devices.

TABLE II: Baseline Kernels Under Comparison

Kernel	Parallelism	Bitwidth	Mapping
SOAP3-dp [39]	inter-query	2 bits	one-to-one
CUSHAW2-GPU [45]	inter-query	2 bits	one-to-many
NVBIO [3]	inter-query	2,4,8 bits	one-to-many
GASAL2 [9]	inter-query	4 bits	one-to-one
SW# [35]	INTRA-query	8 bits	one-to-many
ADEPT [13]	INTRA-query	8 bits	one-to-one

*(All kernels have been modified to have at least four-bit GPU-assisted packing and to support one-to-one mapping.)

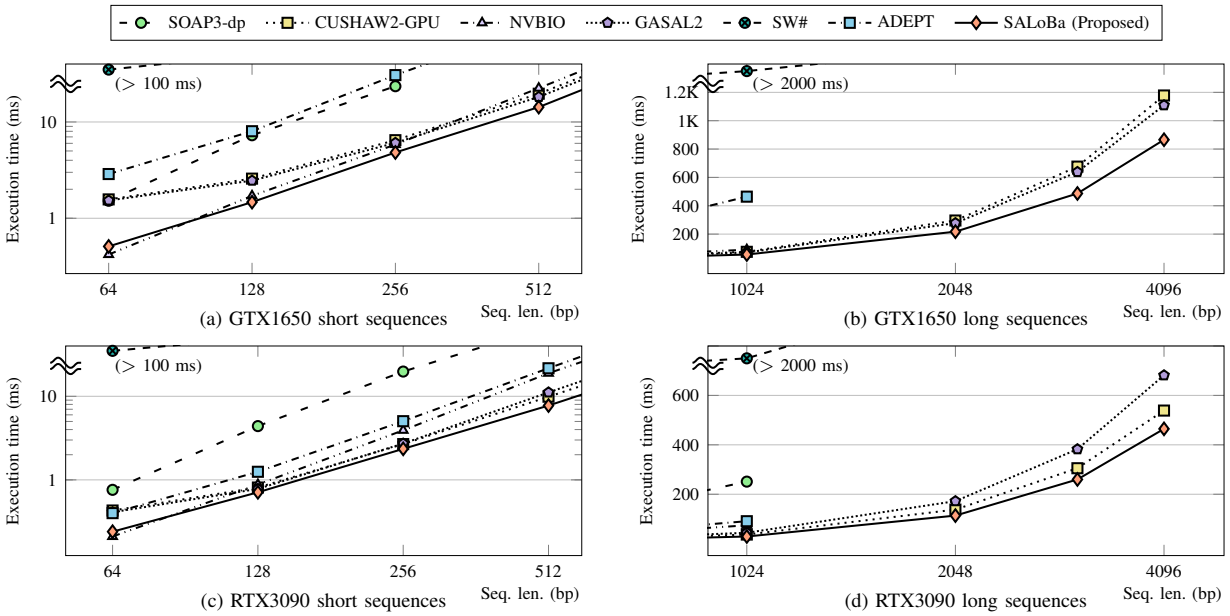


Fig. 6: Performance comparison between extension kernels. The y axes of the (a) and (c) are in log scale for better visibility.

For a fair comparison, we have put our best efforts into optimizing the existing methods under the same environment. Most importantly, we assume on-GPU sequence packing for all methods. GASAL2 is known to achieve state-of-the-art performance using a custom sequence packing kernel executed on GPUs. While the strategy is successful, the packing is orthogonal to the seed extension itself and the same method can be applied to all other kernels. We have modified SOAP3-dp, CUSHAW2-GPU, and ADEPT kernels to support five possible types of literals by unifying the kernels to work on four-bit packing. We left SW# to use its original eight-bit packing, because modifying it to support packing would complicate the memory access behavior. In addition, we have modified some kernels (CUSHAW2-GPU, SW#, and NVBIO) that only have one-to-many alignment modes to support one-to-one alignment.

B. Kernel Performance Measurements

In this subsection, we compare the performance of the seed extension kernels without load imbalances using input reads of equal lengths. To measure the performance under various input lengths, we used an in-house sequence read simulator similar to Wgsim [7] to generate synthetic reads for each length in the range of 64 to 4096. Each kernel processed 5,000 reads per call 200 times, and these results were averaged to output the results in Fig. 6. The execution times were measured with the $cudaEventElapsedTime()$ API.

Fig. 6 (a) and (b) present the performance comparison of the methods on the GTX1650 card, and (c) and (d) present that on the RTX3090 card. For lengths equal to or longer than 128 bp, SALoBa outperforms all other methods. For a very short length of 64 bp, the execution time of NVBIO is slightly shorter than SALoBa (0.42 ms vs 0.51 ms in GTX 1650 and 0.21 ms vs 0.24 ms in RTX 3090). This is reasonable because with intra-query parallelism, the effect of resource under-utilization from prologue and epilogue is relatively significant. However, the break-even point is found at 128 bp, where the reduced global memory access of SALoBa becomes dominant. The methods other than SALoBa that utilize intra-query parallelism (SW# and ADEPT) perform poorly compared to the others. SW# is especially slow because it divides a single DP table into multiple kernel calls where one kernel call processes an anti-diagonal group, resulting in very low resource utilization.

A rather surprising observation is that GASAL2 does not always show superior performance over other baselines. While GASAL2 reports multi-fold speedup compared to its previous work, its speedup is mainly from the on-GPU packing, not the extension strategy. Because we combined CUSHAW2-GPU with the on-GPU packing module taken from GASAL2, it shows comparable results on GTX1650 for all lengths and slightly better performance on RTX3090 for long sequence lengths.

The best speedup obtained at short lengths by SALoBa is observed at 512 bp. The speedup is 27.7% on GTX1650 and 43.6% on RTX3090 against GASAL2. At longer sequence lengths, some baseline kernels fail to run due to structural limitation (ADEPT) or bounded device memory (NVBIO and SOAP3-dp). The performance trend becomes more consistent for longer lengths because the portion of the overhead of global memory access in execution time diminishes. Another cause for this trend would be the decrease of the relative size of prologue/epilogue compared to the sequence length. For inputs equal to or longer than 1,024 bp, the speedup of SALoBa against GASAL2 is consistently around 30% on GTX 1650 and 50% on RTX 3090. Against CUSHAW2-GPU, the

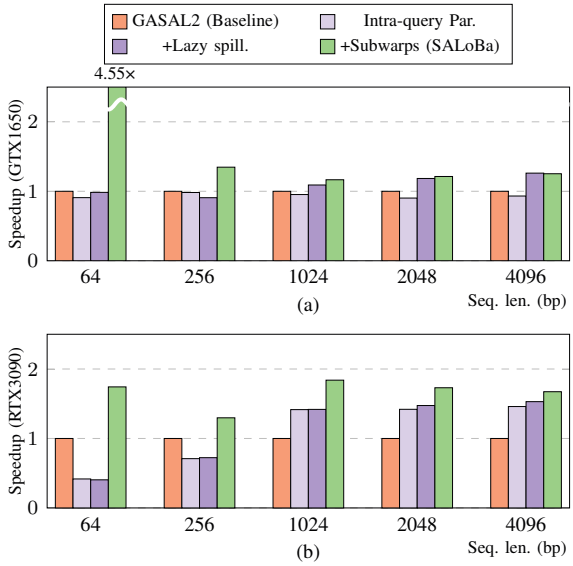


Fig. 7: Ablation study for (a) GTX1650 and (b) RTX3090.

speedup is around 40% on GTX1650 and 20% on RTX 3090.

C. Ablation Study

Fig. 7 breaks down the contributions of the three techniques composing SALoBa. The speedups are normalized against GASAL2, which performs reasonably well along all sequence lengths we target. The most effective technique for shorter lengths ($\leq$1024) is subwarp scheduling, which is expected because at shorter inputs, the portions of the prologue and the epilogue are very large. Therefore, switching to intra-query parallelism yields performance degradation. Using subwarp scheduling directly reduces the overhead and provides a substantial speedup.

A seemingly large speedup is observed at 64 bp for both GTX1650 and RTX3090, which is counter-intuitive because there is less intra-query parallelism that SALoBa can exploit. However, further analyzing Fig. 6 (a) and (c) reveals that it is the inefficiency of GASAL2, not speedup of SALoBa. GASAL2 has a relatively large memory initialization cost at the beginning, and using a small sequence length fails to amortize it. Compared to NVBIO instead of GASAL2 at 64 bp, it correctly reflects the intuition that inter-query parallelism is better suited for very short queries.

As the input becomes longer, the gain from using subwarps becomes marginal, and the effects of the other two techniques become significant. Intra-query parallelism and lazy spilling both contribute to less global memory access. The former reduces the amount of data stored in the global memory, whereas the latter reduces redundant access by coalescing access better. Therefore, these become the main driver for performance gains in longer input ranges. Interestingly, in RTX3090, intra-query parallelism has more effect, whereas lazy spilling is more effective for GTX1650. We believe this is explainable by the fact that RTX3090 has a higher computation/memory bandwidth ratio. RTX3090 has a peak performance of 35.58 TFlops, and its memory bandwidth is 936.2 GB/s using GDDR6X. On the other hand, GTX has a peak performance of 2.98 TFlops with 128.1 GB/s memory bandwidth. The ratio is 38.91 Flops/B in RTX3090 and 23.82 Flops/B in GTX1650. This means that RTX3090 is generally more bottlenecked at its memory. The major drawback of intra-query parallelism is the low CUDA thread utilization. However, as RTX3090 is more bottlenecked in memory bandwidth, it is likely that there are more computational resources to process the data even when some are being idle. This partially offsets the utilization problem in RTX3090.

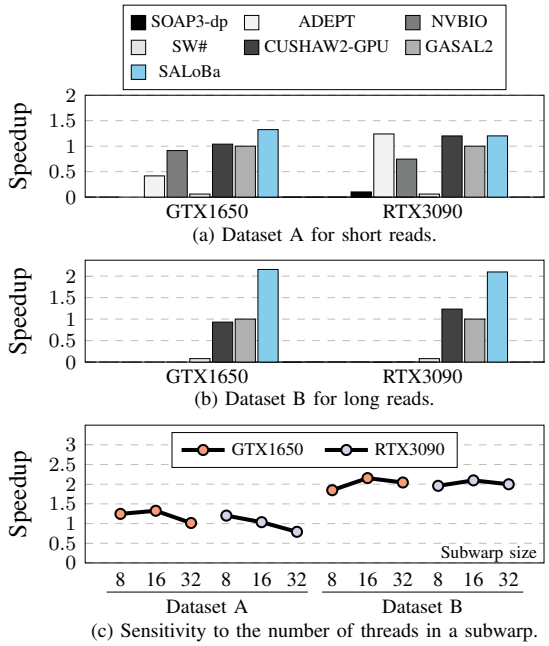
D. Real-world Data Experiments

To test SALoBa under workload imbalance with the realistic distribution of workload sizes, we generate seeds from BWA-MEM [37] using real-world datasets. It takes a whole reference genome sequence and many sequence reads to generate seeds that are composed of multiple pairs that can be processed by extension kernels.

For the reference genome sequence, we used the latest release of the human genome assembly project, Build 38 patch release 13 (GRCh38.p13) [2] that has 3.1G base pairs (bp). For the input sequence reads, we used two datasets downloaded from the sequence read archive [36]. The **dataset A** (SRR835433) is from a 2nd generation sequencer Illumina MiSeq and represents short reads where each sequence has a length of 250 bp. The dataset comprises 8.3M sequences which are randomly read genome parts from a human. Each sequence is read twice, resulting in the total number of base pairs to be 250$\times$8.3M$\times$2=4.1 Gbp. The **dataset B** (SRP091981) is from a 3rd generation sequencer PacBio RS and represents long reads. It also contains randomly read genome parts from a human where there are 82K sequences with variable lengths that averages around 2,000 bp to a total of 182.4 Mbp. The distributions of the seeds processed by BWA-MEM [37] are presented in Fig. 2.

The results are plotted in Fig. 8. Fig. 8 (a) shows the performance for short read dataset A. The best speedup of SALoBa over the baseline GASAL2 is 32.5% for GTX1650 and 20.2% for RTX3090. SOAP3-dp could not complete the workload on GTX1650 as some of the inputs exceeded the length it could process. The speedup values observed from SALoBa are slightly larger than that of Fig. 6 (a) and (c) due to the fact that SALoBa suffers less from workload imbalance. While the performance of SOAP3-dp and NVBIO are inferior to GASAL2, CUSHAW2-GPU exhibits some speedup over GASAL2 (with the help of GASAL2's on-GPU packing kernel). However, its speedup is smaller than that of SALoBa. ADEPT achieves a similar speedup compared to SALoBa only on RTX3090, but the speedup comes from placing all the intermediate values in the shared memory (no global memory access), which fundamentally limits ADEPT up to 1024 bp.

An interesting difference between GTX1650 and RTX3090 is that the optimal subwarp size was 16 for GTX1650 and eight for RTX3090 (see Fig. 8 (c)). This outcome is due to



(a) Dataset A for short reads.
(b) Dataset B for long reads.
(c) Sensitivity to the number of threads in a subwarp.

Fig. 8: Performance of SALoBa on a real-world data. The speedup is normalized to that of GASAL2.

the different effectiveness of each technique. According to Fig. 7, the benefit of applying subwarp scheduling for shorter sequences was 2.26× for GTX1650 and 2.85× for RTX3090 in geomean. In GTX1650, the benefit of using subwarps is partially offset by the overhead from workload imbalance, shifting the optimal configuration to 16 threads per subwarp.

Fig. 8 (b) reveals the performance for dataset B with longer reads. On this dataset, SOAP3-dp, ADEPT, and NVBIO fail to run due to their input length limitations. The speedup of SALoBa greatly improves compared to that of Fig. 6 (b) and (d) because the amount of workload imbalance increases which works in favor of SALoBa. The best speedup is around 2.1× for both GTX1650 and RTX3090. Subwarps with 16 threads per subwarps gained the best performance, which is the sweet spot between exploiting intra-query parallelism and reducing the workload imbalance.

VI. Related Work

A. GPU-based Smith–Waterman Algorithm Implementations

Accelerating Smith–Waterman algorithm [57] with GPUs has been studied for many years. Some earlier work based on OpenGL library [40], [41] addressed issues for mapping the algorithm on graphics pipeline. A popular CUDA implementation can be found from the CUDASW++ family [42], [47], [49]. They suggest utilizing both the intra- and inter-query parallelism based on a pre-determined sequence length threshold [42]. Similar approaches can be found from GSWABE [46] and gpu-pairalign [14]. In addition, focused on all-to-all patterns for protein DB alignments, query profiling optimization [47] and CPU-GPU hybrid parallelism [49] have been suggested. However, such optimizations are often too specific to protein alignments, and cannot be easily generalized to other domains such as DNAs.

In such regard, the CUDAlign family [20], [21], [52], [54] focus on a general Smith–Waterman algorithm using intra-query parallelism. CUDAlign [54] splits the DP table into multiple blocks and distributes them to threadblocks. Then, the communication between the threadblocks are done using a dedicated region in the global memory called the 'horizontal bus' and 'vertical bus'. The algorithm is later extended to support linear space algorithm to reduce the memory size [52]. Its recent versions support multi-GPU alignment with execution time prediction [21] and traceback [20]. When the DP table is too large, approaches such as MultiBP [18], MASA [19], and SW# [35] suggest block pruning [53] that removes some blocks that can never achieve the optimal score.

However, these algorithms are optimized for huge sequences that sometimes do not even fit into a single GPU card [33]. For DNA alignment softwares with seed-and-extend strategies, the input sequence length for the extension ranges around several hundreds even with the long DNA reads. Because of this, libraries that rely on inter-query parallelism [3], [9] often perform much better for DNA alignments as demonstrated in [13]. On the other hand, SALoBa outperforms the previous approaches on DNA alignment scenarios by adopting several careful optimizations.

B. GPU-accelerated Sequence Alignment Softwares

Some approaches design an end-to-end DNA alignment software that are friendly to GPU architectures. SARUMAN [15] and GPU-RMAP [11] are some early methods that implement hashtable lookups on GPUs to perform short read mappings. With the increased use of the Burrows–Wheeler transform (BWT) [38] based on suffix-trees, BarraCUDA [34], SOAP3 [39] and CUSHAW1 [48] were introduced with GPU implementations of BWT. SOAP3-dp [50], CUSHAW2-GPU [45], and LOGAN [60] are later methods that adopt seed-and-extend strategy, and Arioc [59] further expands the strategy to multiple GPUs. However, these methods often sacrifice alignment quality for the sake of throughput. For example, SOAP3 and LOGAN only supports limited number of mismatches, and CUSHAW2-GPU packs the sequences to two bits by converting the fifth 'N' bases to random bases.

Although the approaches above show good throughput with only a small amount of quality degradation, the industry has grown to value quality over speed. As high-quality algorithms such as BWA-MEM [37] became a de facto standard, the usability of the GPU-aware alignment softwares were limited. Some approaches [13], [29]–[31] tackle this problem and design a seed extension kernel general enough to be used for BWA-MEM using intra-query performance. However, later approaches based on inter-query parallelism outperformed these kernels, which is the strategy adopted by the current state-of-the-art methods such as NVBIO [3] or GASAL2 [9].

Finally, it is worth mentioning cuBLASTP [61], a GPU accelerated protein search algorithm that uses a variant of seed-and-extend strategy. However, cuBLASTP only accelerates the

seeding part using GPU, and leaves the extension part to CPU for utilizing CPU-GPU hybrid parallelism.

C. FPGA/ASIC Acceleration

As interest in the genomics pipeline has increased, more researchers have considered designing dedicated accelerators using FPGAs or ASICs. [62] provided a implementation of the Smith–Waterman algorithm on an Altera FPGA, followed by a few other studies using banded algorithms [16], [26] or flexible systolic arrays [10], [28]. Moreover, DRAGEN [56] is an FPGA-based platform from Illumina that implements a full end-to-end analysis into an FPGA. Darwin [58] is an ASIC accelerator that speeds up the whole genome sequencing through the co-design of both the software and hardware, powered by gapped filtering. GenAx [22] is an automata-based accelerator for both seeding and extension.

Although these approaches provide significant speedup, FPGAs or ASICs are much harder to design and it requires a long time to reach the market. Compared to them, GPU-based alignment software could be a quick solution easily adopted by any system that has GPUs attached.

VII. Discussion

A. CUDA Shuffle Instructions

The CUDA shuffle instruction is an alternative to shared memory by allowing a direct register-to-register data exchange for inter–thread communication. However, using shuffle-based communication did not add any additional speedup on top of SALoBa. This outcome aligns with the CUDA specification that the throughput of the shuffle instructions is almost the same as the nonconflicting shared memory reads [8].

One potential gain from shuffle instructions is reducing shared memory usage. However, the portion of data cached in the shared memory was negligible in the current scenario and did not provide noticeable speedup.

B. Banded Algorithms

Many seed extension methods use the banded algorithm to reduce the computational burden. Taking advantage of the fact that the best matching path of the DP table is usually near the diagonal, processing cells within some predetermined width (the *band*) often yields solutions of sufficient quality. However, most GPU-based extension libraries do not exploit this.

One problem is that the band sizes for each query are often different, which worsens load balancing. However, we envision that banded algorithms would have more benefits and could be adopted for better performance with longer reads.

C. Multiple GPUs

It is often beneficial to use multiple GPUs, especially when equipped on a single machine. We believe that extending this work by splitting the queries into equal numbers and assigning them to multiple GPUs would be straightforward. One possible drawback of such a strategy would be the load imbalance between the GPUs. However, the penalty would be small compared to the thread-level imbalance problem. If this becomes significant, it could be solved by dynamic assignment or preprocessing with approximate sorting.

VIII. Conclusion

We proposed SALoBa, a new GPU-based seed extension library, exhibiting state-of-the-art performance over the existing work. The experiments reveal that the software performs well on modern devices, and the performance gain increases with recent GPUs. We believe this software will be useful in fields where sequence alignment times are critical for diagnosing fatal diseases.

Acknowledgements

This work has been supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2022R1C1C1008131, 2022R1C1C1011307) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2020-0-01361, Artificial Intelligence Graduate School Program (Yonsei University), 2021-0-00853, Developing Software Platform for Programming of PIM).

References

[1] "DNA sequencing costs: Data." [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data
[2] "GRCh38.p13." [Online]. Available: https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.39/
[3] "NVBIO." [Online]. Available: https://developer.nvidia.com/nvbio
[4] "NVIDIA Tesla P100." [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[5] "NVIDIA TESLA V100 GPU ARCHITECTURE." [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[6] "NVIDIA TURING GPU ARCHITECTURE." [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
[7] "Reads simulator." [Online]. Available: https://github.com/lh3/wgsim
[8] "Throughput of Native Arithmetic Instructions." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions__throughput-native-arithmetic-instructions
[9] 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, pp. 1–20, 2019.
[10] 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*. IEEE, 2015.
[11] A. M. Aji, L. Zhang, and W.-c. Feng, "GPU-RMAP: Accelerating short-read mapping on graphics processors," in *CSE*. IEEE, 2010.
[12] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.
[13] M. G. Awan, J. Deslippe, A. Buluc, O. Selvitopi, S. Hofmeyr, L. Oliker, and K. Yelick, "ADEPT: A domain independent sequence alignment strategy for GPU architectures," *BMC bioinformatics*, vol. 21, no. 1, pp. 1–29, 2020.
[14] J. Blazewicz, W. Frohmberg, M. Kierzynka, E. Pesch, and P. Wojciechowski, "Protein alignment algorithms with an efficient backtracking routine on multiple GPUs," *BMC bioinformatics*, vol. 12, no. 1, pp. 1–17, 2011.
[15] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short-read alignment to microbial genomes using graphics processing unit programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–1358, 2011.
[16] P. Chen, C. Wang, X. Li, and X. Zhou, "Hardware acceleration for the banded Smith-Waterman algorithm with the cycled systolic array," in *FPT*. IEEE, 2013.

[17] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, "The A100 datacenter GPU and ampere architecture," in *ISSCC*, vol. 64. IEEE, 2021, pp. 48–50.

[18] M. A. C. de Figueiredojr, J. P. Navarro, E. F. de Oliveira Sandes, G. Teodoro, and A. C. M. Melo, "Parallel fine-grained comparison of long DNA sequences in homogeneous and heterogeneous GPU platforms with pruning," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[19] E. F. De O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. De Melo, "MASA: A multiplatform architecture for sequence aligners with block pruning," *ACM Transactions on Parallel Computing*, vol. 2, no. 4, pp. 1–31, 2016.

[20] E. F. de Oliveira Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. Melo, "CUDAlign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in GPU clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2838–2850, 2016.

[21] F. d. O. Edans, G. Miranda, A. C. de Melo, X. Martorell, and E. Ayguadé, "CUDAlign 3.0: Parallel biological sequence comparison in large GPU clusters," in *CCGrid*. IEEE, 2014.

[22] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "GenAx: A genome sequencing accelerator," in *ISCA*. IEEE, 2018.

[23] 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*. IEEE, 2020.

[24] A. Goyal, H. J. Kwon, K. Lee, R. Garg, S. Y. Yun, Y. H. Kim, S. Lee, and M. S. Lee, "Ultra-fast next generation human genome sequencing data processing using DRAGEN™ bio-IT processor for precision medicine," *Open Journal of Genetics*, vol. 7, no. 1, pp. 9–19, 2017.

[25] 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*. IEEE, 2020.

[26] 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*. IEEE, 2007.

[27] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *PPoPP*, 2011.

[28] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, "An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm," in *SAMOS*. IEEE, 2015.

[29] ——, "GPU-accelerated BWA-MEM genomic mapping algorithm using adaptive load balancing," in *ARCS*. Springer, 2016.

[30] ——, "Hardware acceleration of BWA-MEM genomic short read mapping for longer read lengths," *Computational biology and chemistry*, vol. 75, pp. 54–64, 2018.

[31] E. J. Houtgast, V. Sima, K. Bertels, and Z. Al-Ars, "An efficient GPU accelerated implementation of genomic short read mapping with bwamem," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 38–43, 2017.

[32] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers, "Exploring modern GPU memory system design challenges through accurate modeling," *arXiv preprint arXiv:1810.07269*, 2018.

[33] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010.

[34] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "BarraCUDA-a fast short read sequence aligner using graphics processing units," *BMC research notes*, vol. 5, no. 1, pp. 1–7, 2012.

[35] M. Korpar and M. Šikić, "SW#–GPU-enabled exact alignments on genome scale," *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, 2013.

[36] R. Leinonen, H. Sugawara, M. Shumway, and I. N. S. D. Collaboration, "The sequence read archive," *Nucleic acids research*, vol. 39, no. suppl_1, pp. D19–D21, 2010.

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

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

[39] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao *et al.*, "SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[40] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *IPDPS*. IEEE, 2006.

[41] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, "GPU accelerated Smith-Waterman," in *ICCS*. Springer, 2006.

[42] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC research notes*, vol. 2, no. 1, pp. 1–10, 2009.

[43] Y. Liu, B. Popp, and B. Schmidt, "High-speed and accurate color-space short-read alignment with CUSHAW2," *arXiv preprint arXiv:1304.4766*, 2013.

[44] ——, "CUSHAW3: Sensitive and accurate base-space and color-space short-read alignment with hybrid seeding," *PloS one*, vol. 9, no. 1, p. e86869, 2014.

[45] Y. Liu and B. Schmidt, "CUSHAW2-GPU: Empowering faster gapped short-read alignment using GPU computing," *IEEE Design & Test*, vol. 31, no. 1, pp. 31–39, 2013.

[46] ——, "GSWABE: Faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 958–972, 2015.

[47] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++ 2.0: Enhanced smith-waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC research notes*, vol. 3, no. 1, pp. 1–12, 2010.

[48] ——, "CUSHAW: A CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 2012.

[49] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC bioinformatics*, vol. 14, no. 1, pp. 1–10, 2013.

[50] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung *et al.*, "SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner," *PloS one*, vol. 8, no. 5, p. e65632, 2013.

[51] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[52] E. F. d. O. Sandes and A. C. M. de Melo, "Smith-Waterman alignment of huge sequences with GPU in linear space," in *IPDPS*. IEEE, 2011.

[53] ——, "Retrieving Smith-Waterman alignments with optimizations for megabase biological sequences using GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 1009–1021, 2012.

[54] E. F. O. Sandes and A. C. M. de Melo, "CUDAlign: Using GPU to accelerate the comparison of megabase genomic sequences," in *PPoPP*, 2010.

[55] E. E. Schadt, S. Turner, and A. Kasarskis, "A window into third-generation sequencing," *Human molecular genetics*, vol. 19, no. R2, pp. R227–R240, 2010.

[56] K. Scheffler, S. Kim, V. Jain, J. Yuan, W. Sherman, T. O'Connell, E. Ojard, L. Murray, R. Mehio, and S. Catreux, "Accuracy improvements in somatic whole-genome small-variant calling with the DRAGEN platform," 2020.

[57] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[58] 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*. ACM, 2018.

[59] R. Wilton and A. S. Szalay, "Arioc: High-concurrency short-read alignment on multiple GPUs," *PLoS computational biology*, vol. 16, no. 11, p. e1008383, 2020.

[60] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick, "LOGAN: High-performance GPU-based x-drop long-read alignment," in *IPDPS*. IEEE, 2020.

[61] J. Zhang, H. Wang, and W.-c. Feng, "cuBLASTP: Fine-grained parallelization of protein sequence search on CPU+GPU," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 14, no. 4, pp. 830–843, 2015.

[62] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," in *International Workshop on High-performance Reconfigurable Computing Technology and Applications*, 2007.