Accelerating Smith-Waterman Alignment workload with Scalable Vector Computing

Dong-hyeon Park, Jonathan Beaumont, Trevor Mudge
Computer Science and Engineering
University of Michigan
Ann Arbor, MI
{dohypark, jbbeau, tnm}@umich.edu

***Abstract*—Recent breakthroughs in DNA sequencing opened up new avenues for bioinformatics, and we have seen increasing demand to make such advanced biomedical technologies cheaper and more accessible. Sequence alignment, the process of matching two gene fragments, is a major bottleneck in Whole Genome Sequencing (WGS). We explored the potential of accelerating Smith-Waterman sequence alignment algorithm through ARM's scalable vector extension (SVE) as an alternative to scalar and traditional SIMD implementations using three vectorization paradigms: batching, slicing, and Wavefront. Our SVE implementation is able to achieve over 18× speedup over a NEON implementation and up to 54× speedup over a scalar CPU implementation for long queries, with minimal memory bandwidth overhead that scales efficiently with sequence length.**

***Index Terms*—Accelerator architectures; Genomics; Dynamic programming;**

I. Introduction

Thanks to the wide availability of computation power realized by Moore's law, the field of genome sequencing has seen rapid advancements in the past decade [1]. Gene sequencing is the process of mapping the sequence of base pairs that compose each strand of DNA in an organism. Mapping out the sequences that make up each strand help obtain the genetic code of an individual organism, and provide valuable genetic information for wide array of biological fields, ranging from forensics to biotechnology. The original cost of sequencing a single genome was $100 million in 2001 when the Human Genome Project was first launched [2]. Since then, the cost of sequencing has been reduced drastically at a rate greater than Moore's law, down to only $1,400 in 2015, as shown in Figure 1. These improvements in sequencing cost and time came from applying modern optimization techniques to these sequencing algorithms, as well as the use of GPUs or FPGAs to accelerate the computation [1].

While dedicated FPGA and ASIC platforms can provide significant improvement in performance [3], the rapid advancements in sequencing techniques make it difficult to invest heavily on a static platform. Furthermore, developing user-friendly interfaces and development tools for these specialized platforms incur significant resource overhead. Although application-specific platforms can provide better maximum throughput, off-the-shelf hardware and general-purpose systems provide greater degree of flexibility and accessibility that can be valuable in such a rapidly evolving field.

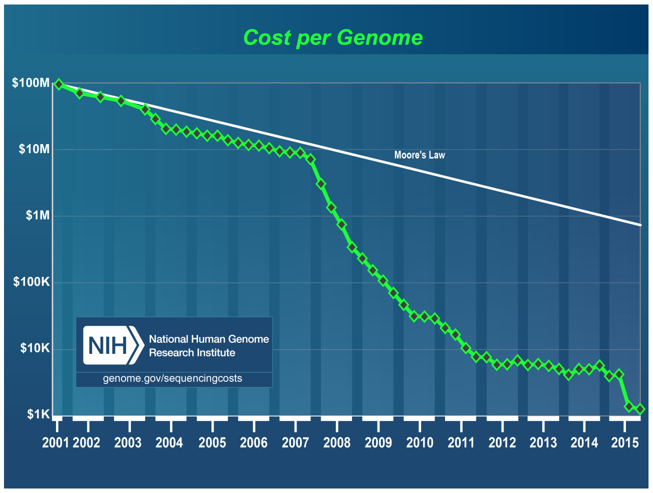


Figure 1: Cost of Sequencing a Genome (National Human Genome Research Institute [2])

This paper explores the benefit of using SIMD architectures to accelerate sequence alignment workloads. We show that the data-dependent nature of the algorithms do not map well to traditional SIMD architectures such as ARM's NEON architecture. Accordingly, we focus on accelerating sequence alignment through the use of ARM's new Scalable Vector Extension (SVE) architecture and explore the benefits provided by vector-length agnostic code in fine-tuning sequencing workloads, as well as other features better suited for scientific computing.

The rest of the paper is organzied as follows. Section II gives a background on genomic sequencing and details previous work on this topic. Section III describes the algorithm we are accelerating in detail, and we illustrate the acceleration techniques we employ in section IV. Section V explains our methodology, and section VI evaluates our results. We conclude with a summary of future work to be done in section VII.

II. Related Works

A. Whole Genome Sequencing

Whole genome sequencing is the process of determining the complete DNA sequence, and it is a major part of computational biology. The gene sequencing pipeline is often broken into four primary steps: read, align, assemble, and analysis [4]. In the read phase, short fragments of gene sequences are sampled from the target organism through a sensor. Signal or image processing techniques are used on sensor data to obtain the protein codes of the sampled fragments.

In the alignment phase, the short sampled fragments are compared the reference genes in the known database using sequence alignment algorithms. Sequence alignment algorithms detect similar regions between two sequences and produce a similarity score for assessing how close they are. Smith-Waterman algorithm [5] is the most well known alignment algorithms, and it used as the basis for most modern alignment tools, such as BLAST [6], Bowtie2 [7], or BWA-SW [8].

The assemble phase uses the sampled fragments to construct the full sequence based on the alignment data. Finally, the analysis phase searches the assembled sequence for variants or abnormalities that are used to diagnose the target organism. Statistical techniques or machine learning algorithms such as deep neural networks and hidden Markov models are commonly used for these analyses.

B. Sequence Alignment

Sequence alignment process takes up a significant portion of a whole genome sequencing pipeline. The open-source genome analysis platform, SpeedSeq [4] takes 13 hours to fully align and annotate a human genomes with 50x coverage, 61.5% (8 hours) of which are spent during the alignment process. There have been several prior works to speedup the Smith-Waterman alignment algorithms in various hardware platforms. SWPS3 [9] accelerates Smith-Waterman through multi-threading and SIMD vector instructions in Intel's x86 or IBM's Cell architecture. CUDASW++ [10] is a CUDA-implementation of the algorithm, and Altera provides an FPGA-implementation for their XD1000 platform [11].

In our study, we investigate the use of ARM's new scalable vector extension (SVE) architecture for accelerating DNA sequencing procedure. While prior works on SIMD-accelerated sequence alignment exists [9], we are the first to explore the impact different SIMD vector lengths have on the performance of these alignment algorithms.

C. Multi-core Systems and Near Memory Computing

As more cores get packed onto a single chip, large many core systems are expected to dominate the future processor market. While many core systems are effective in speeding up compute-intensive, parallel workloads, many modern applications are limited by the memory bandwidth of a system. In our initial investigation of genomics workload, we explored the use of multicore systems as a means to speedup DNA sequencing, as the sequencing workload is known to exhibit high level of parallelism as shown in Figure 2. In addition, the use of Processor-In-Memory (PIM) systems on 3D stacked memory is a potential solution to alleviate memory bandwidth bottlenecks. However, our SIMD optimization manages to resolve the concerns of memory utilization in a more efficient manner.

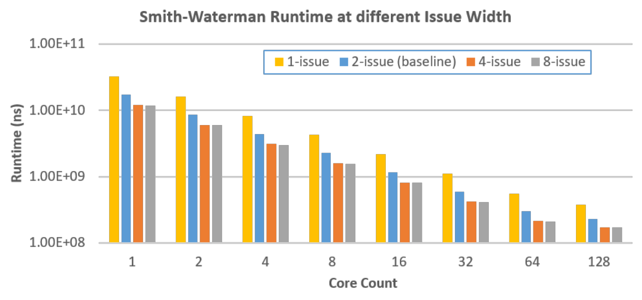


Figure 2: Smith-Waterman Alignment in Multi-core system

D. Scalable Vector Extension (SVE)

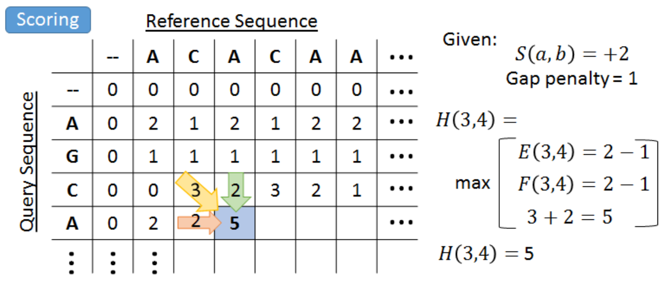
Scalable Vector Extension (SVE) is a new vector architecture by ARM, designed to complement their existing SIMD architecture. Unlike existing SIMD architectures such as ARM's Advanced SIMD (more commonly known as NEON [12]), SVE proposes a flexible SIMD ISA that is vector-length agnostic: the same binary code can be run across hardware implementations with different data widths. The SVE makes it easier for programmers to scale their code to larger machines and choose the optimal data width for their workload and system parameters without needing to rewrite the code for different vector lengths. SVE architecture comes with its own 32 vector registers, as well as 16 predicate registers that can configure and mask the SIMD pipeline. The logical and arithmetic SVE operations are executed in the SVE hardware pipeline. In the baseline implementation of SVE, the memory instructions are broken down into regular ARM micro-ops and processed by the host processor's memory unit, but more advanced implementations can improve vector memory operations through memory coalescing.

III. Smith-Waterman Algorithm

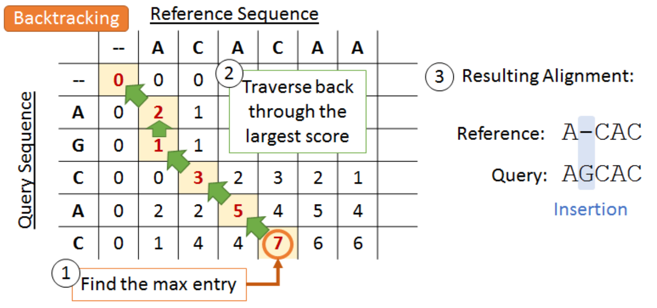
Smith-Waterman is a dynamic programming algorithm that produces local, pairwise alignment between two string sequences. Smith-Waterman takes two string sequences that are in FASTA format: a reference sequence and a query sequence. The reference sequence is the baseline sequence that comes from the gene database. The query sequence is the sampled fragment that needs to be aligned to the reference. Typical length of the query sequence range from 30 base-pairs (bps) to 400 bps, depending on the technology used for sampling the sequence. Smith-Waterman outputs the segment of the two sequences that has the best match and the alignment score that indicates how similar the two segments are. The alignment process has two main parts to the alignment process: generating the scoring matrix, and back-traversing the largest score.

A. Scoring Stage

The scoring matrix is a dynamically generated m+1 by n+1 matrix, where m is the length of the reference sequence and



(a) Scoring Stage



(b) Backtracking Stage

Figure 3: Smith-Waterman Alignment Overview

n is the length of the query sequence, as shown in Figure 3a. The first row and column of the matrix are initialized to zero, and the matrix is filled by traversing through the matrix from the top-left corner.

Given two sequences a and b, the score of each matrix, $H(m, n)$ is calculated as shown:

$$H(m,n) = max \begin{cases} E(m,n) \\ F(m,n) \\ H(m-1,n-1) + S(a_m, b_n) \end{cases}$$

$$E(m,n) = max \begin{cases} H(m,n-1) - g_o \\ E(m,n-1) - g_e \end{cases}$$

$$F(m,n) = max \begin{cases} H(m-1,n) - g_o \\ F(m-1,n) - g_e \end{cases}$$

The constants, g_o and g_e, are gap penalties that compensate for insertions or deletions in the sequence. $S(a_m, b_n)$ is the substitution matrix that describes the rate in which amino acid denoted by a_m transitions to b_n.

To calculate the score at position (m, n), values from three adjacent position needs to be compared: horizontal $(m-1, n)$, vertical $(m, n-1)$, and diagonal $(m-1, n-1)$. The $E(m, n)$ and $F(m, n)$ values derived from the horizontal and vertical positions represent a misalignment that skips a base in the query or the reference sequence. In both instances, the values coming from adjacent entries are subtracted by the gap penalty to take compensate for the mismatch. The direct one-to-one alignment is represented by the diagonal value, $H(m-1, n-1) + S(a_m, b_n)$.

B. *Backtracking Stage*

Once the full matrix is created, the algorithm finds the entry with the largest score, as shown in Figure 2. Starting from this maximum entry, the algorithm looks at top top, left or top-left adjacent entries and tracks back to the one with the largest score. A diagonal movement indicates a match, horizontal movement implies a deletion, and vertical movement denotes a insertion on the query sequence. The process is repeated until a cell with zero value is reached, and the resulting continuous path describes the location and characteristic of the most optimal alignment discovered by the algorithm.

IV. Accelerating Smith-Waterman Algorithm

The majority of the computation time in Smith-Waterman algorithm is spent generating the scoring matrix. For a reference sequence of length m and query sequence of n, the matrix computation time scales by $O(mn)$. We accelerated the scoring matrix computation in vector architecture using three different methods: batch, sliced, and Wavefront.

A. *Batch Smith-Waterman*

The batch implementation accelerates the sequence alignment process by operating on multiple reference-query pairs at once. Given a pool of query sequences that need to be aligned, the algorithm distributes a query to each vector as shown in Figure 4, and forms a batch of queries to be processed together. The number of reference-query pairs that the SIMD unit can process depends on the width of the vector unit. All the vectors operate synchronously, but do not have any data dependencies between each other. This optimization focuses on improving the throughput, while keeping the processing latency bound by the largest length query sequence in the batch.

B. *Sliced Smith-Waterman*

The sliced algorithm (or more commonly referred to as ”Striped” [13]) accelerates the Smith-Waterman algorithm itself by vectorizing parts of score matrix computation. The reference sequence is partitioned by the number of vectors available, and each vector operates only on its partitioned segment, as shown in Figure 5.

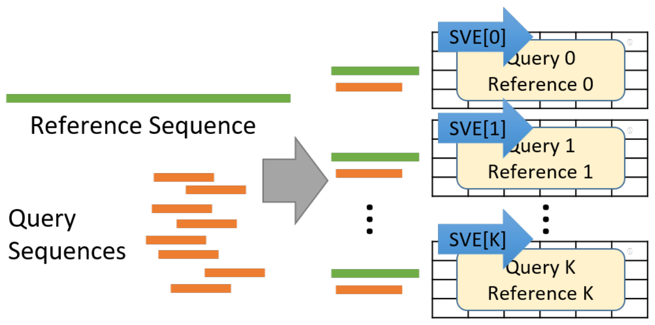


Figure 4: Batch Smith-Waterman Overview

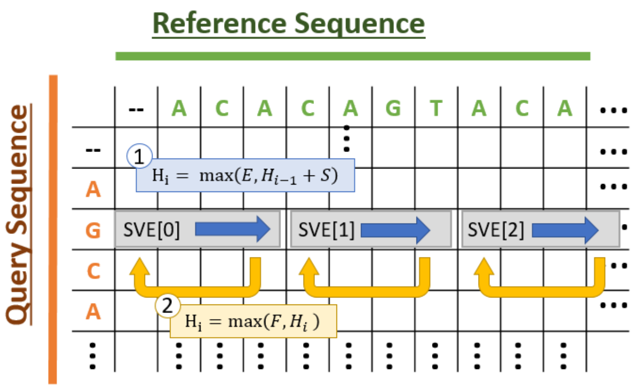


Figure 5: Sliced Smith-Waterman Overview

The operations conducted by each vector slice is shown in Algorithm 1. For each row m of the matrix computation, the scores with vertical ($E(m,n)$) and diagonal ($H(m-1,n-1)$) data dependencies are calculated first. Then the algorithm makes a second pass through the row to resolve the horizontal data dependency, $F(m,n)$. During this second pass, the algorithm checks whether the horizontal component $F(m,n)$ is bigger than the similarity score of current entry.

If the horizontal component is smaller, then all the subsequent scores do not need to be updated because the horizontal dependency has no impact on the future entries. Therefore, the horizontal update can be bypassed, reducing the overall computation by a significant amount for long sequences that resolve the dependency early. The slicing help maximize the data reuse for a given row of score matrix at the cost of increase in worst-case computation introduced by the additional pass through the row [5].

C. *Wavefront Smith-Waterman*

The Wavefront implementation traverses the similarity matrix diagonally, so that each iteration depends solely on the results from the previous iterations [14]. The algorithm starts from the top-left entry in the top-left corner of the similarity matrix, and travels across the matrix with each iteration until

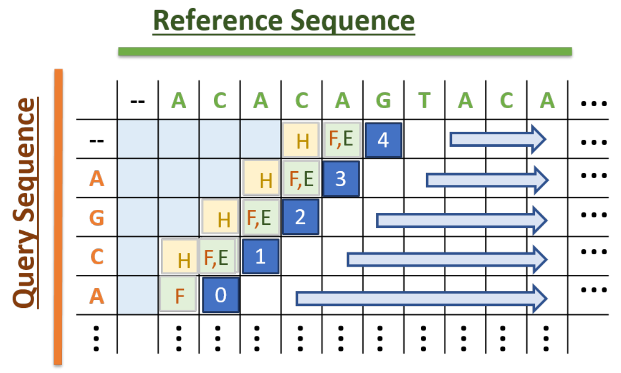


Figure 6: Wavefront Smith-Waterman Overview

Algorithm 1 Sliced Smith-Waterman

Precondition: r and q are sequences of length M and N. L is the vector length.

```
1:  function SLICEDSW($x, y$)
2:      $H \leftarrow$ M-by-N score matrix
3:      $Hmax \leftarrow 0$
4:      $E, F, E_{prev} \leftarrow 0$ : vectors of length $N$
5:      for $i \leftarrow 0$ to $M$ do
6:          for each vector: $k \leftarrow 0$ to $L-1$ do:
7:              for $j \leftarrow k$ to $k + N/L$ do
8:                  $E[j] \leftarrow \max(H_{i-1}[j]$ - $g_o$, $E_{prev}[j]$-$g_e)$
9:                  $F[j] \leftarrow \max(H_i[j-1]$ - $g_o$, $F[j-1] - g_e)$
10:                 Htemp $\leftarrow H_{i-1}[j-1] + S(r[i], q[j])$
11:                 Htemp $\leftarrow \max(E[j]$, Htemp)
12:                 $H_i[j] \leftarrow \max(F[j]$, Htemp)
13:                 $E_{prev}[j] \leftarrow E[j]$
14:         Hrmax $\leftarrow 0$ : vector of length $L$
15:         for each vector: $k \leftarrow 0$ to $L-1$ do:
16:             for $j \leftarrow k$ to $k + N/L$ do
17:                 if $(H_i[j-1] - g_o > F[j-1] - g_e) \forall$ vectors
    then
18:                     Exit to outer most loop (advance $i$)
19:                 $F[j] \leftarrow \max(H[j-1]$ - $g_o$, $F[j-1] - g_e)$
20:                 $H_i[j] \leftarrow \max(F[j], H_i[j])$
21:                 Hrmax$[j] \leftarrow \max(H_{i-1}[j]$, Hrmax$[j])$
22:         Hmax$[i] \leftarrow$ vmax(Hrmax)
23:     return $H$, Hmax
```

the whole matrix is filled. Given the entry m in the n-th diagonal, the horizontal $F(m,n)$ and vertical $E(m,n)$ dependencies come from the entries in the $n-1$-th diagonal, and the diagonal dependency $H(m-1,n-1)$ comes from the $n-2$-th diagonal as shown in Figure 6.

At each iteration, the diagonal strip of entries keeps growing by one until the length is equal to the length of the smaller of the two input sequences. Each diagonal strip is partitioned by the vector width. The vectors will be underutilized during the early stages of the execution due to the number of operating entries being less than the available vector length, but lanes will remain at high utilization for majority of execution time for long input sequences.

The Wavefront algorithm resolves the data dependencies of the Smith-Waterman algorithm by scheduling the execution along the diagonal so only the entries that have the all the available dependencies are executed. In return, the addressing of the memory operations are more fine grained than the continuous or strided accesses of bulk and sliced implementations.

V. Experimental Setup

Our study was conducted using a custom version of gem5 with SVE simulation provided by ARM. We modeled a out-of-order 64-bit ARM core with 8-wide issue and 1GHz clock as our baseline system. The fixed-vector SIMD architecture

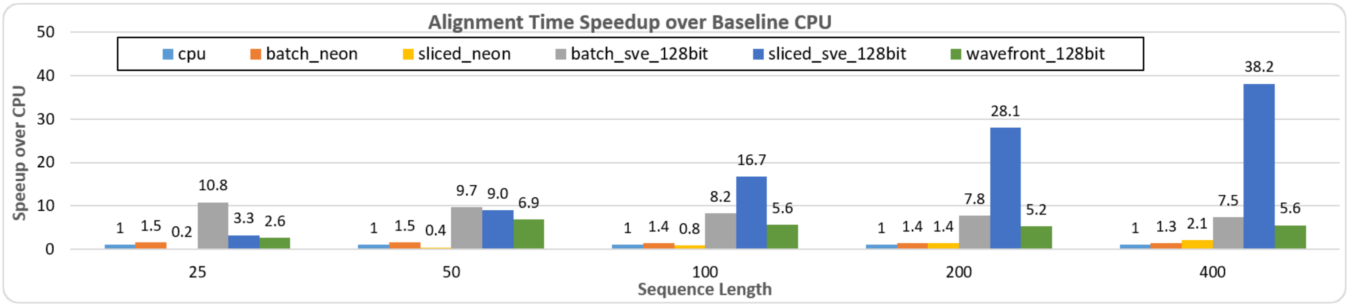


Figure 7: Sequencing Speedup of Different Algorithms compared to Baseline CPU with 64kB L1 Data Cache. Vector widths of SVE implementations are set to equal to that of NEON machines (128-bit). Batch performs the best under short sequences, while Sliced is optimal for long sequences. Wavefront performs poorly at 128-bit, because it requires wide vector lengths to achieve better performance. Note that the size of each individual vectors in Batch, Sliced, and Wavefront algorithms were 32-bit, 16-bit, and 64-bit respectively.

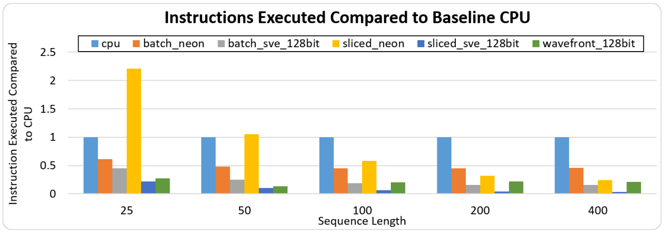


Figure 8: Instructions Executed during sequencing over Baseline CPU with 64kB L1 Data Cache.

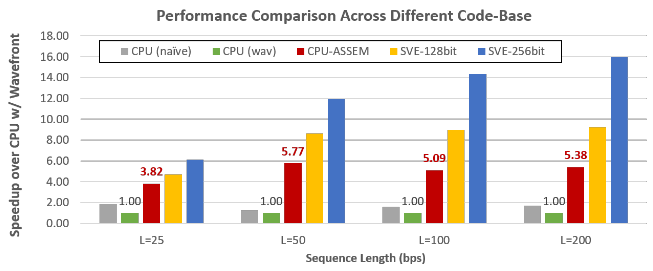


Figure 9: Performance speedup over Wavefront algorithm written in C. CPU (naive) is standard Smith-Waterman algorithm in C, while CPU (wav) is single-scalar Wavefront in C. CPU-ASSEM is Wavefront algorithm written in assembly without any SVE instructions. SVE-128bit and SVE-256bit are Wavefront algorithm with SVE in assembly, running at 128-bit and 256-bit vector widths respectively.

(NEON) has data width of 128-bits, while the SVE architecture has four different data widths ranging from 128-bits to 1024-bits. Due to the lack of an available C compiler for SVE, the SVE implementation was hand-written in assembly code. The detailed simulation specification is given in Table I.

The sample query and reference sequences conducted for testing were taken from a database of *E.Coli 536* strain from NCBI Database. The 4.9 million base pairs long strain was randomly sampled into smaller reference segments of length 25, 50, 100, 200, and 400 bps for collecting performance data. The query samples were obtained through the use of Wgsim sequence read simulator [15].

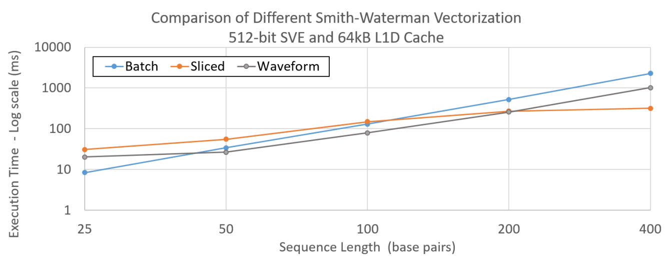
Table I: Gem5 Simulation Specification

Component	Configuration
Core	Single-Core out-of-order 64-bit ARM, 1GHz, 8-issue SIMD Width: 128-bit (NEON), 128-1024-bit (SVE)
Cache	32KB private L1 instruction cache, 2-way associative 64KB private L1 data cache, 2-way associative 4MB private L2 inclusive cache, 8-way associative
DRAM	Capacity: 8GB Latency: 30 ns Memory Controller Bandwidth: 12.8 GB/s
Benchmark	Batch and Sliced Smith-Waterman Reference: 25-400bp sections from *E.Coli 536* Strain Query: 1000x 25-400bp

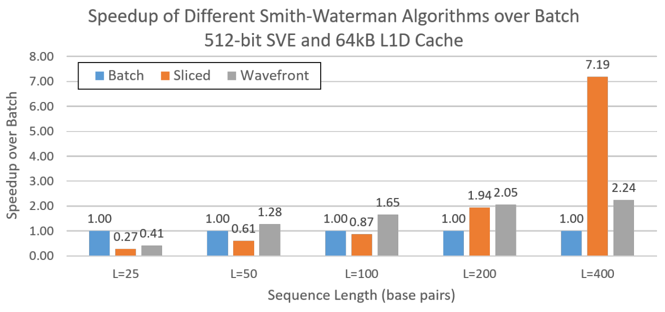
VI. Experimental Analysis

A. *Performance of SVE over Existing SIMD Architecture*

The speedup of total execution time of aligning 1000 query sequence to a reference sequence of similar size was plotted in Figure 7. The baseline serial implementation of Smith-Waterman algorithm that is organized in the similar manner as the Batch Smith-Waterman algorithm if the vector length was 1. The NEON implementations of Batch and Sliced use the NEON instructions to accelerate the score matrix creation, and utilize its 128-bit SIMD units. While CPU and NEON code were written and compiled in C, the SVE implementations of Batch, Sliced and Wavefront, were written in assembly by calling the SVE instructions directly. The SVE code makes use of SVE's vectorized load and store instructions, as well as horizontal reductions and predicated instructions which are not available in NEON instruction set. The SVE configuration shown in Figure 7 is 128-bit to match the data width of NEON. For Batch, each base-pair was represented in 32-bit, Sliced was represented in 16-bit, while base-pairs in Wavefront were represented in 64-bit due to the limitations in vectorized memory operations of SVE that were required to execute each

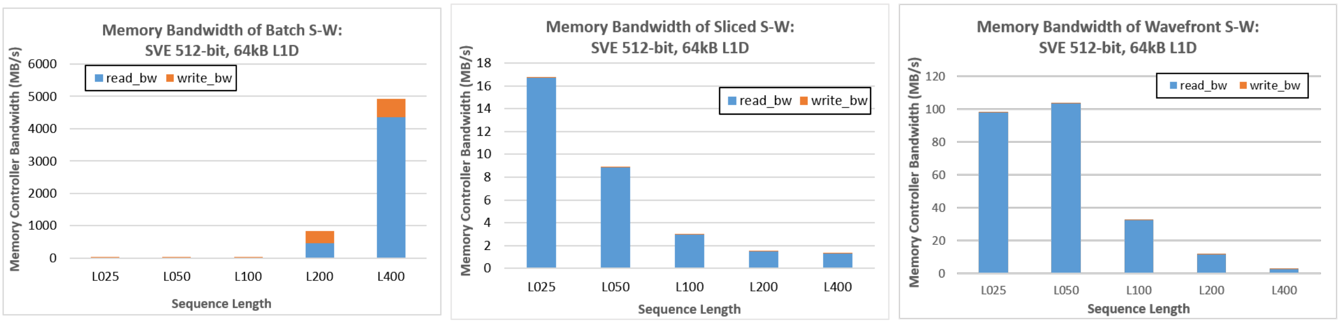


(a) Runtime Comparison



(b) Speedup over Batch

Figure 10: Performance Comparison of Different Smith-Waterman for SVE 512-bit with 64KB L1 Data Cache System. Sequencing time is the total time required for aligning 1000 queries. Batch is favored for sequences below 50 bps, Wavefront is favored between 50-200 bps, and Sliced is favored for sequences 400 and longer.



(a) Batch Memory Bandwidth (b) Sliced Memory Bandwidth (c) Wavefront Memory Bandwidth

Figure 11: Average Read and Write Memory Bandwidth of Smith-Waterman for Batch, Sliced, and Wavefront Optimization. Maximum bandwidth of the System is $12.8GB/s$. Sliced and Wavefront makes more efficient use of memory bandwidth because they operate only on a single sequence pair at a time.

algorithm efficiently.

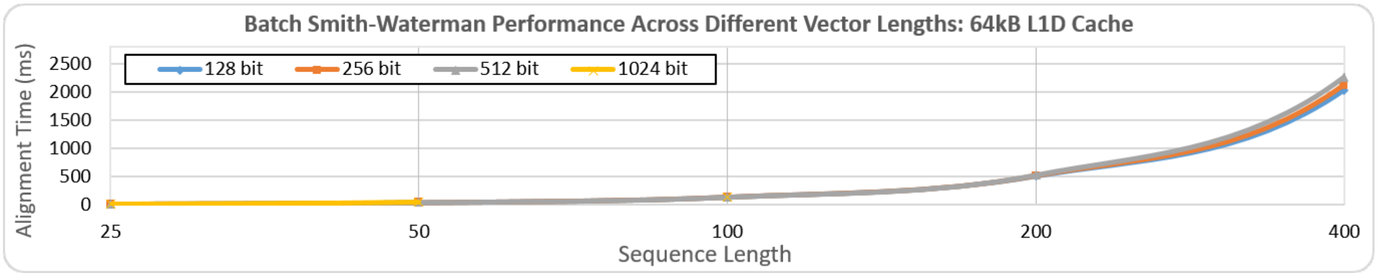
The SVE configurations of the three algorithms outperform the serial implementation by 3-38x at all sequence lengths, and outperform the NEON implementations. The advantage of SVE code over NEON code comes from more efficient utilization of vector registers through predication, and significant reduction in number of executed instructions as shown in Figure 8. SVE version is able to utilize more advanced vector instructions such as predicated execution, horizontal max operation, and vector memory addressing which are absent in NEON ISA.

Lastly, efficiency of the hand-written assembly code was a significant factor in the performance advantage of SVE over NEON or CPU. As shown in Figure 9, the waveform implementation written in assembly without any SVE instructions resulted in 4-6x speedup over the same algorithm written in C.

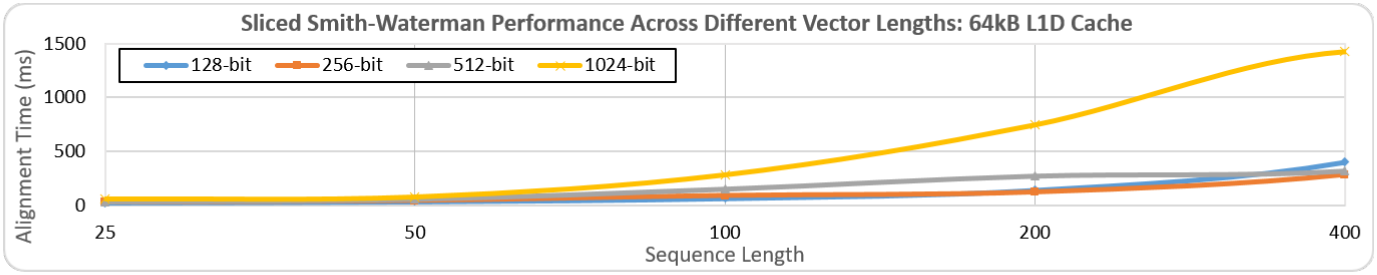
B. Comparison of Different Algorithms

The performance of Batch, Sliced, and Wavefront Smith-Waterman are compared in Figure 10. The sequencing times are taken from SVE 512-bit configuration with 64kB of L1 Data cache. The Batch Smith-Waterman outperforms both Sliced and Wavefront solutions for queries smaller than 50 base pairs. Wavefront algorithm exhibits better speedup between sequences of length 50 and 200, while Sliced Smith-Waterman starts to outperform the rest heavily beyond 400 base pairs. The simplicity and low overhead of the Batch implementation help the algorithm perform favorably for short sequences, while Wavefront implementation performs favorably at medium-length sequences due to its efficient utilization of the SIMD hardware.

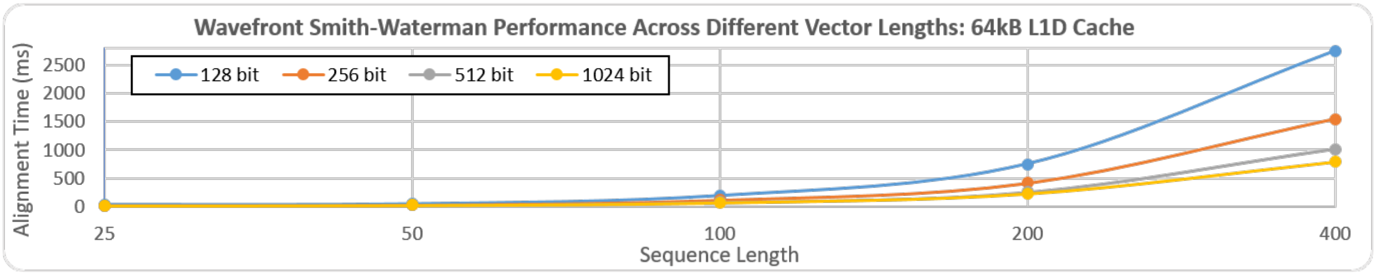
The Sliced Smith-Waterman performs poorly at low sequence lengths due to the high overhead of execution, but scales favorably at larger sequence lengths due to the bypassing of the horizontal dependency calculation. At low sequence lengths, Sliced Smith-Waterman only has a limited window where bypassing helps reduce the computation overhead of the algorithm. At short sequence lengths, Sliced Smith-Waterman can perform worse than even a naïve CPU implementation, due to the performance overhead of resolving horizontal dependency by traversing the same row twice. But at longer sequences, it is more likely for the bypassing to be triggered early on in the re-computation step, resulting in significant amount of computation savings. This is also the reason Sliced algorithm struggles to take advantage of the wider vectors. Wider vectors help the Sliced algorithm compute the initial



(a) Batch Performance



(b) Sliced Performance



(c) Wavefront Performance

Figure 12: Performance of Smith-Waterman for Batch, Sliced and Wavefront Optimization. Runtime is the total time it takes to align 1000 query sequences to reference sequence of similar size. Vector width has minimum impact on the performance of Batch, and Sliced exhibit worse performance for wider vectors at short sequences. Wavefront algorithm is takes advantage of the wide vector length with diminishing return for each additional lane.

calculations for diagonal and vertical dependencies faster, but the increased segmentation reduces the bypassing window, resulting in a performance decrease for short and medium sized sequences (less than 200 bps).

The memory bandwidth utilization shows the strength of Sliced and Wavefront implementation in minimizing the memory overhead. Figure 11 show the average read and write memory bandwidth of the three implementations at different sequence lengths. The memory bandwidth demands of Batch algorithm increase dramatically with larger sequence sizes and start to become a performance bottleneck. On the other hand, Sliced and Wavefront implementations manage to reduce the memory bandwidth with increasing sequence length. This is because both Sliced and Wavefront Smith-Waterman implementations only operate on a single query at a time and the dependent data required to perform matrix calculation are only accessed once. Improvements in data reuse help minimize bandwidth overhead in Sliced and Wavefront optimization, and lead to improved performance. Comparing Sliced and Wavefront algorithms, Sliced algorithm require significantly less memory bandwidth compared to the Wavefront algorithm. This is because Sliced algorithm accesses the sequences more efficiently by operating on contiguous blocks of memory during its execution, while Wavefront algorithm makes heavy use of fine-grained data accesses, causing the memory controller to request multiple cache lines at ones where only a few portion of the data was actually requested.

C. Vector Width Variation

The sequencing time of Batch, Sliced, and Wavefront implementations at different SVE vector widths ranging from 128-bit to 1024-bit are shown in Figure 12. Batch Smith-Waterman shows an exponential increase in alignment time with increasing sequence length for all vector widths, and the execution time remains relatively the same regardless of vector width, with larger vectors performing worse under most circumstances. This is due to algorithm's high dependence on vector memory operations, which are not improved with vector width as SVE simply breaks down vector loads and stores into individual micro ops. On the other hand, alignment time

Table II: Optimal Algorithm and Vector Width Selection at Each Sequence Length

Sequence Length	Algorithm	SVE Width	Speedup over CPU	Speedup over Sliced 256-bit
25 bps	Batch	128-bit	10.8	3.7
50 bps	Wavefront	1024-bit	12.0	1.7
100 bps	Sliced	128-bit	16.7	1.4
200 bps	Sliced	256-bit	32.5	1.0
400 bps	Sliced	256-bit	53.9	1.0

of Sliced algorithm does not grow as rapidly as the Batch solution. While large SVE vectors performs poorly for small length queries, the wide 512-bit vector do not degrade in performance as rapidly as the smallest 128-bit configuration.

Unlike Batch and Sliced implementations, the Wavefront algorithm exhibits strong performance scaling with increase in vector length, especially for long sequences. Doubling the vector width from 128-bit to 256-bit leads to 1.3-1.8x speedup depending on the sequence length. While the performance gain shows a diminishing return with each additional SIMD lane, the Wavefront algorithm takes full advantage of the additional hardware, thanks to the algorithm's handling of data dependency in its execution.

The optimal algorithm and SVE vector width combination at each sequence length is shown in Table II. For large sequence lengths, our solution is able to achieve up to 53.9x speedup over the CPU solution. We show that dynamically selecting the algorithm and SIMD width depending on the input sequence length leads to significant performance speedup over a homogeneous solution. The vector agnostic design of SVE architecture makes it an optimal platform for such dynamic scheduling solutions.

VII. Future Work

For future work, we will explore accelerating other sequence alignment algorithms such as Burrow-Wheeler alignment (BWA) using SVE. Furthermore, we plan to integrate the Burrow-Wheeler alignment to a workload scheduler that selects between Batch, Sliced, and Wavefront algorithm depending on the length of the alignment region selected by Burrow-Wheeler algorithm. We expect the hybrid system to achieve better performance and accuracy than existing solutions, as shown in Table II by the variation in optimal algorithm and hardware combinations at different sequence lengths.

Gene assembly workloads are another area of computational biology that can be explored. De novo assembly [16] is the process of assembling a genome from the spliced fragments without relying on the alignment to a reference gene. Such assemblers relies on large de Bruijin graphs to complete their operation, and the graph computation capabilities of Processor-In-Memory (PIM) has potential to accelerate such workloads greatly.

VIII. Conclusion

The rapid advancements in computational biology has made bioinformatic more accessible than ever before. Faster and more efficient sequencing hardware will continue making DNA sequencing cheaper and more ubiquitous, opening up new opportunities for their use in forensics, medical diagnosis, and even biological computing. Here, we analyzed the characteristics of Smith-Waterman algorithm and its performance on SIMD platforms, particularly ARM's new SVE architecture. Smith-Waterman algorithm showed great speedup with our SVE optimizations, achieving up to 53.9× speedup over baseline CPU implementation given optimal selection of algorithm and hardware. We developed three vector implementations of Smith-Waterman and categorize the different performance characteristics with respect to sequence length and vector configuration.

References

[1] S. Tsoka and C. a. Ouzounis, "Recent developments and future directions in computational genomics." *FEBS letters*, vol. 480, no. 1, pp. 42–48, 2000.

[2] "DNA Sequencing Costs: Data." [Online]. Available: https://www.genome.gov/sequencingcostsdata/

[3] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *IEEE Design and Test*, 2014.

[4] C. Chiang, R. M. Layer, G. G. Faust, M. R. Lindberg, D. B. Rose, E. P. Garrison, G. T. Marth, A. R. Quinlan, and I. M. Hall, "SpeedSeq: ultra-fast personal genome analysis and interpretation," *Nature Methods*, vol. 12, no. November 2014, pp. 1–5, 2015. [Online]. Available: http://www.nature.com/doifinder/10.1038/nmeth.3505

[5] M. Zhao, W. P. Lee, E. P. Garrison, and G. T. Marth, "SSW library: An SIMD Smith-Waterman C/C++ library for use in genomic applications," *PLoS ONE*, 2013.

[6] W. J. Kent, "BLAT - The BLAST-like alignment tool," *Genome Research*, vol. 12, no. 4, pp. 656–664, 2002.

[7] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nat Methods*, vol. 9, no. 4, pp. 357–359, 2012.

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

[9] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz, "SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2." *BMC research notes*, vol. 1, no. 1, p. 107, 2008. [Online]. Available: http://bmcresnotes.biomedcentral.com/articles/10.1186/1756-0500-1-107

[10] 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, p. 117, 2013. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3637623/

[11] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications held in conjunction with SC07 - HPRCTA '07*, p. 39, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1328554.1328565

[12] "Introducing NEON^{TM} Development," *ARM Limited*, 2009.

[13] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007. [Online]. Available: + http://dx.doi.org/10.1093/bioinformatics/btl582

[14] S. Steinfadt, "SWAMP+: Enhanced Smith-Waterman Search for Parallel Models," *ICPPW*, 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6337464/

[15] H. Li. (2011) WGSIM - Read Sequence Simulator. [Online]. Available: https://github.com/lh3/wgsim/

[16] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean, "De novo assembly and genotyping of variants using colored de Bruijn graphs," *Nature Genetics*, vol. 44, no. 2, pp. 226–232, 2012.