

A Review of the Smith-Waterman GPU Landscape

Richard Barnes



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-152

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-152.html>

August 13, 2020

Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Richard Barnes was supported by the Department of Energy's Computational Science Graduate Fellowship (Grant DE-FG02-97ER25308) and, through the Berkeley Institute for Data Science, by the Gordon and Betty Moore Foundation (Grant GBMF3834) and by the Alfred P. Sloan Foundation (Grant 2013-10-27).

Tests were performed on the Titan and Summitdev supercomputers managed by Oak Ridge National Laboratory's Leadership Computing Facility; the Cori supercomputer managed by the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231; and XSEDE's Comet supercomputer, which is supported by the NSF (Grant ACI-1053575) and was available to RB through the NSF Graduate Research Fellowship.

A Review of the Smith-Waterman GPU Landscape

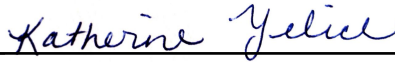
Richard Barnes

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Katherine Yelick
Research Advisor

8/13/2020

(Date)



Adjunct Assistant Professor Aydın Buluç
Second Reader

A REVIEW OF THE SMITH-WATERMAN GPU LANDSCAPE

RICHARD BARNES^{A,*}

^a*Electrical Engineering & Computer Sciences, Berkeley, USA
Energy & Resources Group, Berkeley, USA
Berkeley Institute for Data Science, Berkeley, USA*

Abstract

A key step in the assembly of genomes is the identification of locally optimal alignments between small subsections of the genome. The Smith-Waterman algorithm provides an exact solution to this problem at the cost of significantly greater computation versus approximate methods. The need to advance both the speed and sensitivity of local alignment has driven a great deal of research on accelerating the Smith-Waterman algorithm using GPUs, which we review here. We find that some optimization techniques are widespread and clearly beneficial, while others are not yet well-explored. We also identify a limited set of algorithmic motifs which can be used to classify all of the existing Smith-Waterman GPU implementations. This exposes gaps in the literature which can be filled through future research.

1. Introduction

Knowing the sequence of the human genome facilitates a better understanding of disease processes [56] while knowing the genomes of agricultural crops, such as rice, facilitate efforts to improve their resilience and yield [61]. Metagenomics, the study of genes within an environment, can help us understand how varying microbial populations in the human gut affect health [60] and how varying populations affect ecosystems and ecosystem services [26]. However, to obtain these benefits it is necessary to be able to assemble the genetic data.

Genome and metagenome assembly is a process in which a genome is cut into small pieces so it can be processed into digital forms. Once digitized, these many small segments of DNA (reads) are assembled into longer sequences (contigs) in a process analogous to assembling a puzzle without knowing what the final picture looks like. Proteins, while generally shorter, are assembled via a similar process. If two sequences overlap with high certainty, then they can be joined into a single, longer sequence. The process of determining this overlap is *local sequence alignment*. Finding overlap candidates nominally involves comparing all sequences against all

other sequences to find the highest scoring overlaps. Metagenome assembly problems can have up to 50 million sequences or more, making a direct all-pairs comparison infeasible. Instead, graph algorithms and heuristics are used to reduce the size of the search space and parallel techniques are used to reduce wall-times [4, 16]. Still, after this extensive filtering, many pairwise sequence alignments may still need to be done.

The Smith-Waterman algorithm, detailed below, is used to find the optimal alignment between two sequences in $\Omega(mn)$ time, where m and n are the lengths of the sequences. To avoid this cost, heuristic algorithms such as X-drop [94] and BLAST [85] are used. Even these heuristic algorithms can be expensive; for instance, the SeqAn X-drop aligner [14] takes 90% of the compute time when used to bootstrap from reads to contigs in the BELLA long-read aligner [23]. While optimal alignments may desirable, it's clear that they must be calculated very efficiently to be competitive with heuristic methods.

In response to this need, Smith-Waterman has been repeatedly implemented to leverage low-level aspects of target hardware including CPU SIMD instructions [96], graphics GPUs [50], general-purpose compute GPUs [1], the Sony Playstation [91], FPGAs [64, 80], and theoretical work has

*Corresponding author. ORCID: 0000-0002-0204-6040
Email address: richard.barnes@berkeley.edu
(Richard Barnes)

been on quantum computing algorithms [58]. Here, we review these implementations and show that, perhaps surprisingly, despite the amount of research that has been done on mapping Smith-Waterman to hardware, the design space has still only been shallowly explored and common design flaws in existing research products limit their utility. In response, we develop a reusable library to solve some common design issues. Since quantum computers and FPGAs are not yet widely available, we focus the review on CPU and GPU implementations, particularly the latter.

2. The Smith-Waterman Algorithm

The Smith-Waterman algorithm [78] computes the optimal local alignment of two sequences of length m and n . How the SW algorithm is instantiated depends heavily on both the *gap penalty* as well as a substitution function. We review several gap penalties below, but note that only the constant, linear, and affine penalties have seen significant optimization work on GPUs.

A potential alignment between two DNA or amino acid sequences is given positive scores if characters between the sequences match, and negative scores if they don't (§ 3). However, if a sequence has gained or lost many characters in a row (often in a single biological event), this may result in many mismatches, indicating that the potential alignment is highly unlikely. Such "gaps" can be detected and assigned a gap penalty which penalizes the gap at a lower rate than the sequential mismatches, giving a better model of the biology.

As inputs, the algorithm takes

- Two sequences $A = a_1, a_2, \dots, a_m$ and $B = b_1, b_2, \dots, b_n$
- A weight matrix $W(a_i, b_j)$ where $W \leq 0$ if $a_i \neq b_j$ and $W > 0$ if $a_i = b_j$
- Gap penalties $G < 0$ which are incurred for either initiating or extending a gap

2.1. Constant/Linear Gap Penalty

For a constant gap penalty, the algorithm operates in $O(mn)$ time. For $1 \leq i \leq m$ and $1 \leq j \leq n$ the value is given by:

$$H_{i,j} = \max \left\{ \begin{array}{l} 0 \\ H_{i-1,j} + g \\ H_{i,j-1} + g \\ H_{i-1,j-1} + W(a_i, b_j) \end{array} \right\} \quad (1)$$

If $i < 1$ or $j < 1$, then $H_{i,j} = 0$. Since each update to the matrix depends only on adjacent entries, the space required is proportional to the length of the longest antidiagonal and is therefore $O(\min(m, n))$.

2.2. Affine Gap Penalty

Since, biologically speaking, multicharacter gaps can be created by a single evolutionary event, it makes sense to weight the opening of a gap more heavily than its extension. An affine gap penalty—that is, a penalty in which the cost of a gap of length k has the form $G(k) = uk + v$ with $u, v \geq 0$ —achieves this.

While the original Smith-Waterman algorithm [78] allowed deletions or insertions (gaps) of any length with an affine penalty, it did so at the cost of $O(m^2n)$ time. A modification by Gotoh reduced this to $O(mn)$ time [20]. The space requirement is again $O(\min(m, n))$, proportional to the longest antidiagonal.

For an affine gap penalty, the inputs to the algorithm are largely the same as for the constant gap penalty, though the penalty is now specified by

- A penalty G_{init} for starting a gap
- A penalty G_{ext} for extending a gap

We now introduce two matrices E and F which give the alignment scores for ending with a gap along A and B .

$$E_{i,j} = \max \left\{ \begin{array}{l} E_{i,j-1} - G_{\text{ext}} \\ H_{i,j-1} - G_{\text{init}} \end{array} \right\} \quad (2)$$

$$F_{i,j} = \max \left\{ \begin{array}{l} F_{i-1,j} - G_{\text{ext}} \\ H_{i-1,j} - G_{\text{init}} \end{array} \right\} \quad (3)$$

An insertion to one sequence can always be seen as a deletion to the other sequence. However, with the current setup E covers the case where there is an extra character in A to account for, so we don't advance the pointer to B and instead pay the cost of inserting a character into A . F covers deletion: B has an extra character so we don't advance A and pay the cost of a deletion. While the costs for insertion and deletion could be made asymmetrical, this isn't common in biology.

Given the above, the alignment score is given by:

$$H_{i,j} = \max \left\{ \begin{array}{l} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + W(a_i, b_j) \end{array} \right\} \quad (4)$$

If $i < 1$ or $j < 1$, then $H_{i,j} = E_{i,j} = F_{i,j} = 0$.

2.3. Double affine gap penalty

Introduced by Liu et al. [41] the penalty has the form

$$G(n) = a + \min(n, k)b_0 + \max(0, n - k)b_1 \quad (5)$$

for a gap size of n and generalizes the standard affine gap penalty by imposing a separate penalty for each gap space beyond a given threshold. Few details are available on the algorithm or its implementation.

2.4. Double affine gap penalty (another one)

Gotoh [21] notes that DNA and RNA codons are three-character sequences and therefore, gaps which are not multiples of three should have additional penalties applied to discourage, but allow, frameshifts in the genetic code. An algorithm is available [21], but describing it here is beyond the scope of this paper.

2.5. Logarithmic gap penalty

Human and rodent DNA have power-law distributions of gap sizes which suggests that a logarithmic penalty of the form

$$G(n) = a + b \ln n \quad (6)$$

where n is the gap length may be appropriate [5, 22]. This penalty and its complexity are discussed in further detail below (see § 2.6 and § 2.8).

2.6. Log-affine gap penalty

Cartwright [8] compared the performance of the affine and logarithmic gap penalties, as well as a log-affine gap penalty

$$G(n) = a + bn + c \ln n \quad (7)$$

and concluded that log-affine gap penalties produced the best accuracy followed closely by affine gap penalties; logarithmic gap penalties produced poor alignments. Cartwright concluded that although log-affine gap penalties were better than affine gap penalties, the difference may not be great enough to justify the extra computation time (see § 2.8).

2.7. General gap penalty

The general gap penalty takes the same inputs as before, but makes no assumptions about $G(n)$. Let the value of a particular i, j be given by $V(i, j)$. For $i < 1$ or $j < 1$, $V(i, j) = 0$; otherwise,

$$V(i, j) = \max \left\{ \begin{array}{l} 0 \\ V(i-1, j-1) + W(a_i, b_j) \\ \max_{0 \leq k \leq j-1} (V(i, k) - G(j-k)) \\ \max_{0 \leq k \leq i-1} (V(k, j) - G(i-k)) \end{array} \right\} \quad (8)$$

Using dynamic programming, the best sequence alignment can be found in $O(nm(n+m))$ time and $O(mn)$ space [83].

2.8. Convex gap penalty

First studied by Waterman [89], the convex gap penalty (which bounds the performance of the logarithmic gap penalty) requires that the penalty satisfy the condition

$$G(N+1) - G(N) \leq G(N) - G(N-1) \quad \forall N \geq 1 \quad (9)$$

That is, the penalty decreases as the gap length increases. While the general algorithm above can solve this problem, convexity allows the inner maximums to be found with binary searches resulting in $O(mn \log(mn))$ time and $O(mn)$ space [19, 49, 83].

3. Substitution Costs

The weight or substitution matrix W in Equations 1, 4, and 8 can take several forms. For protein sequence comparisons a 20x20 matrix is needed. The BLOSUM62 matrix [15, 27] is a common choice among the sequencing algorithms, although PAM [11], JTT [33], and application specific matrices [79] could also be used. Though the original values for the BLOSUM62 matrix were miscalculated, the erroneous values seem to give better search performance [82], although more recent work using different benchmarks disagrees [28]. For DNA, a 4x4 matrix would suffice; however, no sequencing algorithm uses such a matrix. Instead, many algorithms forgo substitution matrices altogether, opting to apply a fixed positive score if two sequence characters match and a negative score otherwise (see Table 2), we refer this to as a match-mismatch (MM) system.

Using fixed substitution costs has computational advantages at the expense of lower predictive accuracy in alignment: the full substitution matrix has 400 elements or 210 if the matrix is symmetric about its diagonal. This is too large to store close to the GPU’s threads, creating bandwidth limitations. Constant memory (§ 4.1) is one way to mitigate this (e.g. [1, 2, 50, 53]), though it is notably unused in NVBIO [55]. Converting an MM design to a full substitution matrix can be difficult. For instance, both NVBIO [55] and GASAL2 [1] pack DNA information into 4-bit nibbles and transact on 32-bit packages containing 8 such units. This increases bandwidth utilization at the expense of packing and unpacking the sequences. Since at least 5 bits are required for storing an amino acid, converting these packages from MM to a full substitution matrix would require rewriting most of the kernels.

4. GPUs

Programming GPUs is challenging due to a complex memory structure coupled with small instruction pipelines and minimal opportunities for caching.

4.1. Memory

GPUs have several kinds of memory of varying sizes and speeds, as listed in Table 1. Making appropriate use of this memory is necessary to obtain good application performance. These memories include:

- **Registers** (per-thread, read-write, fast, very limited size): This memory is visible only to the thread that wrote it and lasts only for the lifetime of that thread
- **Shared memory** (per-block, read-write fast, very limited size): This memory is visible to all of the threads in a block and lasts for the lifetime of the block. This memory facilitates interthread communication and intratask parallelism. Parallel barriers or synchronization points are necessary to prevent race conditions.
- **Local memory** (per-thread, read-write slow, not cached, limited size): Local memory has the same scoping rules as registers do, but it is larger and performs significantly slower. Typically this memory is only encountered when a thread has insufficient registers, in which case it may be used automatically at a high performance penalty.

- **Global memory** (per-grid, read-write, slow, not cached, large size): Any part of the global memory can be read and written by any thread in the grid. In newer GPUs the global memory may exceed 16–32 GB. The optimal access pattern is to have threads in a warp access sequential memory addresses (coalesced accesses); other access patterns can reduce performance by half.
- **Constant memory** (per-grid, read-only, slow/fast, cached, limited size): Typically on the order of 64 kB. If all the threads of a half-warp read from the same address, then accessing constant memory is as fast as a register access; otherwise, it can be very slow. This memory is sometimes used for storing query sequences or substitution matrices [42, 50]
- **Texture memory** (per-grid, read-only, slow but cached, large size): Texture memory is optimized for 1-, 2-, or 3-D spatial locality. Due to the specialized nature of this memory, it is not often used in sequencing.

The memory hierarchy of modern GPUs is complex and the information presented here should be viewed only as an approximation to its usage and structure. Attempts to reverse-engineer the structure and instruction sets of GPUs are available for some varieties including Nvidia’s Volta [32] and Turing [31] architectures.

4.2. Branching

When GPUs reach a branching statement, this serializes execution with those threads that pass the branch condition executing their instructions first and those threads that fail the condition executing second. The result is that conditions can halve performance. More complex conditionals can cause further degradation.

4.3. Page-Locked/Pinned Memory

Operating systems prefer to use RAM as a fast storage bank; however, if RAM becomes full, data can be relocated to disk-storage in swap/virtual memory. Modern hardware allows for memory transfer directly between RAM and a GPU, saving the round-trip time to the CPU, but such transfers are only possible if there is a guarantee that the memory in question will stay in RAM. Applications can allocate page-locked/pinned memory to obtain this guarantee.

Name	Capacity	Cached?	Latency	Bandwidth	Access
Nvidia Quadro GV100 (80 SMs, 1.38 GHz)					
Register	64 KiB	No	Very low		R/W, per-thread
Shared	0–96 KiB	No	Low	12 TiB/s	R/W, per-block
Constant	16–32 GiB	Yes (6 MB L2, 65 KiB L1.5, 2 KiB L1)	Low/High		R, per-grid
Texture	16–32 GiB	Yes (6 MB L2, 32–128 KiB L1)	High	750 GiB/s	R, per-grid
Global	16–32 GiB	Yes (6 MB L2, 32–128 KiB L1)	High	750 GiB/s	R/W, per-grid
Local	16–32 GiB	Yes (6 MB L2, 32–128 KiB L1)	High	750 GiB/s	R/W, per-thread
Nvidia K80 (13 SMs, 875 MHz)					
Register	64 KiB	No	Very low		R/W, per-thread
Shared	0–48 KiB	No	Low	2.5 TiB/s	R/W, per-block
Constant	12 GiB	Yes (1.5 MB L2, 32 KiB L1.5, 2 KiB L1)	Low/High		R, per-grid
Texture	12 GiB	Yes (1.5 MB L2, 16–48 KiB L1)	High	191 GiB/s	R, per-grid
Global	12 GiB	Yes (1.5 MB L2, 16–48 KiB L1)	High	191 GiB/s	R/W, per-grid
Local	12 GiB	Yes (1.5 MB L2, 16–48 KiB L1)	High	191 GiB/s	R/W, per-thread

Table 1: Properties of selected GPUs memories. The data is drawn from a series of microbenchmark surveys [31, 32]. Bandwidth values were measured using benchmarking tools.

4.4. Streams

The performance we extract from a GPU comes from two sources: data parallelism and task parallelism. Data parallelism requires that the same function be run across a dataset. Task parallelism allows us to run different functions on different data. Tasks are executed on separate *streams*. Separate streams can also independently manage data transfer to and from the CPU. The effect of this is that it is possible to hide the latency of data transfer (which is slow) by overlapping it with computation, provided the data being transferred comes to/from *pinned/page-locked memory* (memory which the host operating system is not allowed to move).

In the context of alignment, a frequent optimization is to break the input data into chunks. The chunks are then transferred and processed on separate streams to overlap computation and I/O, as above. Another stream-based optimization, employed by GPU-BSW [92], is to launch forward and reverse kernels for each stream. If the kernels don’t require the same number of threads, this can increase data parallelism by allowing more kernels to be scheduled when fewer threads per kernel are required.

4.5. Latency hiding and data starvation

GPU hardware is optimized for throughput, not latency. That is, on a per-element basis GPUs perform operations slower than CPUs. However, a GPU can operate on many more elements at once. Using multiple streams, as above, helps hide transfer latency. To hide compute latency, the GPU

needs to be given a lot of data. Failing to do so leads to “data starvation”. In § 9 we demonstrate how the GPU implementations can dramatically underperform the CPU if they encounter this condition.

5. Motifs

Algorithms are designed around motifs which incorporate strong assumptions about the nature of the data the algorithms will process. Broadly, these motifs are

- **Single 1:1 Pairwise Alignment (S1:1):** Two sequences being aligned against each other.
- **Multiple 1:1 Pairwise Alignment (M1:1):** Multiple sets of two sequences being aligned against each other.
- **Many-to-One Alignment (M:1):** A large number of sequences being aligned against the same sequence. We only use this to indicate situations where the motif somehow optimizes by performing preprocessing on the reference sequence, aligning multiple queries at once, or otherwise doing something more than just iterating over S1:1 alignments.
- **All-Pairs Alignment (AP):** All the sequences in a set being aligned against all the others. Again, this motif is only used if the structure of the problem is exploited.

The length of the sequences involved in the alignment impacts both the design and performance of the algorithm. Unlike on a CPU, a GPU requires explicit allocation and handling of its many different kinds of memory. This means that choosing

sequence lengths determines how memory can be used; this, in turn, can place limits on performance. For instance, CUDAlign [72] is a S1:1 algorithm in which sequences of 200M+ bases are compared against each other. This requires a GPU cluster. At the opposite end of the spectrum GPU-BSW [92] is an M1:1 algorithm that can only handle sequences less than 1024 bases because it performs the entire alignment within the GPU’s limited shared memory. Assumptions about input data and use cases lead to very different designs. By incorporating length into the broad classifications above, we can identify at least ten possible motifs:

- One Long to One Long (S1:1)
- One Long to One Short (S1:1)
- One Short to One Short (S1:1)
- One Long to Many Long (M:1)
- One Long to Many Short (M:1)
- One Short to Many Short (M:1)
- Many Long to One Short (M:1)
- All-pairs Long to Long (AP)
- All-pairs Long to Short (AP)
- All-pairs Short to Short (AP)
- Many 1:1 Long to Long (M1:1)
- Many 1:1 Long to Short (M1:1)
- Many 1:1 Short to Short (M1:1)

It is possible to build one motif from another; however, there are good reasons to avoid doing so.

5.1. Combining Motifs

In a dataset of sequences that need to be aligned against each other heuristics such as a common k-mer requirements can be used to quickly eliminate some possible alignments. The result is a sparse graph of alignments which need to be performed by SW. If we call the sequences S_1, \dots, S_n , a subset of this graph may look like this, where an X indicates an SW alignment to be performed.

	S_1	S_2	S_3	S_4	S_5	S_6	
S_7	X				X		
S_8		X	X	X			
S_9	X	X	X	X	X	X	(10)
S_{10}		X	X	X			
S_{11}			X				

There are several ways to batch the alignments. One possibility is to treat this as a M1:1 problem; however, this results in limited cache reuse and, for longer sequences, unnecessary memory consumption. An alternative option is to use some of

the other motifs from above in concert: an all-pairs (AP), a many-to-one (M:1), and a many one-to-one (M1:1), like so:

	S_1	S_2	S_3	S_4	S_5	S_6	
S_7	M1:1				M1:1		
S_8		AP	AP	AP			
S_9	M:1	AP	AP	AP	M:1	M:1	
S_{10}		AP	AP	AP			
S_{11}			M1:1				(11)

an alternative arrangement might be:

	S_1	S_2	S_3	S_4	S_5	S_6	
S_7	M1:1				M1:1		
S_8		AP	AP	AP			
S_9	M:1	M:1	M:1	M:1	M:1	M:1	
S_{10}		AP	AP	AP			
S_{11}			M1:1				(12)

It might also be faster to perform everything as an all-pairs operation with extraneous results discarded.

Determining the best batching option is an optimization problem that may require having an appropriate, hardware-specific, cost model with variables for the number and length of the sequences in question. Conceptually, this is similar to the matrix chain multiplication problem: of the $\Omega(4^n/n^{3/2})$ different orders in which matrices can be multiplied, an $O(N \log N)$ algorithm [29] finds the optimal ordering and faster approximate solutions exist [9].

6. Optimizations Employed

In addition to the high-level design choices discussed above, a number of techniques have been used to increase the efficiency of algorithms. We explore these below.

Sorting. A number of the implementations we review (e.g. [42, 45, 50]) sort sequences by length. If the lengths of the sequences vary significantly this helps avoid a situation in which a number of threads are finished and waiting on a single, longer sequence to complete. Our ALBP package (§ 8) includes functions to facilitate such sorting.

Query Profiles. An amino acid substitution matrix has 210–400 elements (§ 3), so if these accesses are relatively random each matrix lookup has a high probability of either missing the L1 cache or triggering a non-coalesced memory access, especially in multi-threaded contexts. If these lookups occur in the inner loop which calculates Equations 1, 4, and 8, then there are $O(nm)$ such misses.

A query profile is generated by iterating along a sequence s and calculating $W(s_i, c)$ for each possible character c which might be paired with s_i . Since the sequence is traversed sequentially and the matrix accesses are non-random, the number of cache misses is reduced to $O(\max(n, m))$, since the query profile need only be calculated for one of the sequences in a pair to be aligned. The gains are magnified if the M:1 motif is used since a single query may be made against potentially millions of reference sequences. In contrast, the cost of constructing the query profile may be too expensive to be worthwhile in the M1:1 case [1]. Different authors have taken a variety of approaches to the data layout for the profile, depending how the parallel techniques used in the alignment (see, e.g., [17, 44, 47, 62]).

Speculation. Farrar [17] and Zhao et al. [96] both assume that the values in the F matrix will only rarely affect the values in the H matrix; that is, that lengthy insertions and deletions are relatively rare. This assumption allows them to parallelize along the matrix rows within an inner loop and correct any mistakes in an outer loop. The effect is that greater parallelism is available throughout the computation. CUDASW++ 2.0 [44] brings this technique to the GPU.

LR-2009 [39] and F-2007 [17] both calculate scores using low-precision, storing information in single bytes using saturating arithmetic. If a value of 255 is reached, then the sequences are aligned a second time using higher-precision. If recalculation is uncommon, this technique can increase parallelism (in SIMD contexts) and increase effective bandwidth.

Search Space Reduction. The long S1:1 motifs require so much compute time that optimizations which might be expensive for small sequences become useful. The CUDASW family [69, 72, 74] makes extensive use of one of these: *block pruning*. The CUDASW family divides large (petacell) matrices into blocks. Conceptually, each block is a mini-SW problem with differing boundary conditions and blocks can be processed in parallel along

a block-anti-diagonal (or in other orientations). If a block can be shown to have such a small score that it’s impossible to give a higher score than one that has already been found, then that block can be ignored, or pruned. The method is described in detail in [69]. The method is formalized mathematically in [75] where it is shown that the method prunes between 0–68.8% of the cells in the matrix depending on (1) the sequences involved, (2) the order in which the H cells are calculated, and (3) the parameter values (e.g. gap extension penalty).

Approximation. For sequences with high similarity, the optimal alignment will fall close to the diagonal of the DP matrix. Formalizing this as an assumption gives the k -banded Smith-Waterman algorithm: only cells within a distance k of the main diagonal are filled. SW# [53] employs this method at scale for long S1:1 alignments, GASAL2 [1] provides it as an option for M1:1 alignments, and NVBIO [55] provides it for M:1 alignments.

It is sometimes the case that progressive misalignments result in the optimal alignment falling outside of the band. At this point, banded-SW cannot recover. Another algorithm, X-Drop [94, 95] provides an alternative heuristic: the alignment only considers cells whose score differs from the current maximum by a value of at most X . This can be thought of as forming a band that moves with the current optimal alignment.

As noted in § 3, using match-mismatch scores instead of full substitution matrices is also a cost-saving approximation.

Space Reduction A naïve traceback algorithm requires $O(nm)$ space. CUDASW [69, 72, 74] and SW# [53] work around this by using the Myers-Miller algorithm [51]. This provides a way of finding tracebacks in global and local alignments in linear space using a divide-and-conquer method. This technique has only used for very long alignments.

Adaptively Algorithm Selection. If sequences are relatively short, they and their intermediate alignment artifacts can fit inside a GPU’s shared memory. As sequences get longer, global memory must be used. Short sequences also offer limited opportunities for parallelism within a DP matrix (but see [92]) versus longer sequences. Hains et al. [25] suggest that a good algorithm will therefore switch between inter-task and intra-task parallelism as the sequence length increases; this optimization

is also used in CUDASW++ [42]. Dicker et al. [12] proposes to use a parallel prefix algorithm [3] to align (globally) sequences of less than 4096 bases and then to switch to a more standard anti-diagonal parallel algorithm for longer sequences. If (long) sequences are within 1% of each other’s lengths, CUDASW++ 3.0 [45] creates a static schedule mapping blocks to processes; otherwise, processes use atomics to claim blocks as their tasks complete.

Packing. Early GPUs had high per-transaction costs to access memory and limited-to-no caching. In response, some algorithms pack four 8-bit integers into a single 32-bit integer [39, 43, 44, 47, 50] to reduce transactions. More recent algorithms do not mention this optimization.

For DNA data, there are four characters, which requires only two bits of storage. Some algorithms also allow for an “N” character symbolizing an unknown base (typically arising from issues with the physical measurement of the sequence to be aligned). Including this character means that at least 3 bits are required, although in practice 4 are used because this increases memory alignment. NVBIO [55] and GASAL2 [1] both pack their data such that one 32-bit integer holds 8 characters. This effectively doubles (global) memory bandwidth at the cost of increased computation; however, since computation is often much faster than memory, the result is a net savings. NVBIO [55] performs sequence packing on the CPU while GASAL2 [1] performs packing on the GPU, which is found to be up to 750x faster.

Sequence packing doesn’t make sense for amino acid data: there are 20 possible bases requiring 5 bits of memory, but 8 are required to get aligned memory accesses.

Wall-time Prediction. High-performance compute environments often require users to estimate the wall-time of their work ahead of time. If the user over-estimates, their job may be scheduled later as it takes time for a large enough block to open up. If the user under-estimates, the job is cut short and all the work may be lost. Similarly, if the user under-estimates the resources they need, the job may not finish in a timely fashion, but if they over-estimate they may be charged an unnecessarily high rate for their compute.

Sandes et al. [74] addresses this for the long S1:1 motif by developing an equation to predict align-

ment wall-time based on the lengths of the sequences and the number of GPUs used. They also develop an equation to predict the speedup if additional GPUs are used, helping users to make informed choices about what resources to request. They show that for the single-GPU case their wall-time prediction error is less than 0.45% and their speedup prediction error is less than 5%.

Constant Memory. Under appropriate conditions (see § 4.1), the GPU’s constant memory can be accessed at the same speed as a register. However, this memory is limited in size (see Table 1). A number of the implementations we review use this memory to store substitution matrices. M:1 aligners also use it to store shorter query sequences.

7. Implementation Review

Table 2 summarizes our review of the existing GPU implementations along with a subset of the CPU implementations. In addition to sequence length and the aforementioned properties—gap penalty (§ 2), motif (§ 5), and substitution matrix (§ 3)—we also consider software engineering aspects of the implementations: whether the code compiles, has tests of its correctness, can work as a library, and runs.

7.1. Compilation

Almost none of the code compiled out of the box. The reasons varied from having to correct hard-coded library paths to having to make extensive changes to the code to upgrade it to newer versions of CUDA or port it to new architectures with different intrinsics (e.g. Altivec). As Table 2 shows, for most of the implementations considered it was possible to get compilation. This often took on the order of minutes to hours, with the notable exception of NVBIO, where the effort took considerably longer due the extensive use of antiquated libraries and low-level hardware features. Even in this complex case, our modifications to NVBIO eventually passed NVBIO’s test suite and we were able to upstream our modifications. This provides some evidence that despite frequent hardware and API changes, CUDA code can be maintained. Compilation issues are detailed in § 15.

7.2. Known to crash?

Perhaps the most basic requirement of code is that it run and that, if it encounters a problem, it notify the user in a meaningful way. Not all of the

packages we consider pass this basic test. A common failure mode was to segfault if the user provided inappropriate command-line arguments. In some cases, there was no documentation as to what those arguments should be. We propose a way to resolve this in § 8. Another common failure mode was for software to optimistically assume the GPU had either infinite memory or infinite threads; that is, the software failed to check that the input sizes were appropriate for the device. This failure mode manifests as CUDA errors. Problematically, these errors look the same whether they arise from poor memory management or other bugs in the software and provide the user no context for their resolution. Therefore, we treat CUDA errors as a crash. Software issues are detailed in § 15.

7.3. Tests

Almost none of the code had any form of testing. Our metric for this was whether the code included unit tests or described an end-to-end testing methodology. Of the code with tests, NVBIO was professionally developed by Nvidia and UGENE has a 12-year development history with 21 contributors. SWIFT also has unit tests while PyPaSWAS has only end-to-end testing.

Unit testing is known to reduce the number of bugs in software. Following the institution of unit testing, Williams et al. [90] found a 20.9% decrease in test defects. If the tests were written incrementally in conjunction with the code itself, as in test-driven development, the decrease is 62–91%. Notably, the developers surveyed felt that unit tests helped them write better code and that unit tests could help them understand the code when they inherit it from others or need to debug it. Similarly Google’s ClusterFuzz tool, which generates randomized inputs into software has identified 16,000 bugs in Chrome and 11,000 bugs spread across 160 open source software packages.

Given this, a lack of unit testing, or reliance on end-to-end testing should undermine confidence in software. This is *especially* true of scientific software, since we are often unsure what the outputs should be. Unit tests are not easily added unless software has been built in a modular way and are most easily added by those familiar with the design of the software, so downstream users are not well-positioned to fill this gap.

As an alternative, end-to-end testing can be used

in conjunction with an implementation which is known to be correct. This may not be possible if sequences are too long to be aligned with other software, as is the case with MASA [71] and CUDALIGN [72], making their lack of unit tests more remarkable. Regardless, end-to-end testing fails to detect if the right answer is obtained for the wrong reasons. As Table 2 shows, many of the implementations do not include a traceback. As a result, the only signal to judge the success of an end-to-end test may be the alignment score; since the same score can appear many times in the H matrix, this is not a good verification. End-to-end testing may also suffer from insufficient sample diversity. For instance, the PyPaSWAS [87] end-to-end test compares only two pairs of sequences. GASAL2 [1] is accompanied by an example dataset containing 20,000 pairs of sequences. These could be converted to an end-to-end test; however, all of the A sequences are 150 base pairs while the target sequences range from 152–277 base pairs. This dataset may be insufficient to detect bugs in shorter or longer sequences. A better end-to-end test would compare sequences with a variety of lengths and bases against each other, similar to our performance comparison in § 9.

It is easy for code to run quickly if it doesn’t need to produce correct results. Race conditions and communication problems underlie many bugs in parallel code. Preventing these classes of bugs often requires synchronization and barriers, both of which can reduce performance, sometimes dramatically. The upshot is that a lack of tests should also undermine confidence in performance results.

A final reason for tests is that they increase performance portability and future-proof software. Without tests, it would have been much more difficult to upgrade NVBIO (see § 7.1) to working state. Having tests allowed us to easily and progressively verify that the changes we made did not affect correctness.

7.4. Is it a library?

Software is useful insofar as it integrates well with other software. In the high-performance context in which GPU aligners are used, this means that useful code will expose an API for integration as a library. Table 2 shows that most of the code we review was not designed with this use-case in mind. Instead, many of the implementations deeply entangle their

user interface with their alignment kernels. The effect is that substantial, error-prone work may be necessary to incorporate the implementations into other projects, reducing their impact and usefulness. SSW [96] is a notable exception and, perhaps as a result, is used in several of the CPU+GPU implementations as the CPU-side aligner.

7.5. Design coverage

As mentioned above, the design of an SW implementation depends on the underlying hardware, choice of gap penalty, motif, substitution matrix, and whether or not tracebacks are calculated. Table 2 shows that nearly all development work has gone into affine gap penalties, leaving obvious room for further work on double affine, logarithmic, convex, and general penalty functions.

Research has been done on both full substitution matrices and match-mismatch penalties, but choices made on this axis represent a point of divergence in the literature. As discussed in § 3, these substitution systems typically don't interconvert well. MM penalties allow for data compression schemes which would need to be rewritten to support full substitution while full substitution has higher computational costs than simple substitution.

While most research has considered many-to-one alignments, the sequence lengths involved are heterogeneous. In early work, query sequences were short—in the hundreds of base pairs and few in number. More recent work has mapped sequences of 100–2000 bases against references of up to a billion bases. For the longer references, a Burrows-Wheeler transform [6] is used as a compression strategy and GPU implementations of this exist (e.g. [55, 93]); NVBIO's BWT claims an 8.5x speed-up versus the BowTie2 [38] CPU implementation.

The one-to-one motif has been explored in three contexts. (1) The naïve SW implementation as well as SIMD implementations such as [17] and [84]. Such implementations can achieve greater parallelism by running multiple threads at once, but the algorithms themselves are designed around pairs of sequences. (2) Single long-sequence comparisons. The long comparison work has been driven by the MASA [71] and CUDAlign [72] package families. Using multi-GPU clusters these compare sequences of up to 200 million bases on matrices of up to

60 petacells relying on a host of techniques including block pruning [75] and speculative fill-in which may be too expensive to apply to shorter sequences. (3) Many short-sequence comparisons.

This final motif is an emerging area of research driven by the desire to leverage GPU-based alignment as an intermediate stage in systems like metagenome sequencers (e.g. DiBella [16] and HipMCL [4]). GASAL2 [1] and GPU-BSW [92] optimize specifically for this use case.

In our review, we did not find any algorithms built around the All-Pairs (AP) motif. This could be because it makes the strongest assumptions about which alignments need to be performed. While an M1:1 aligner can freely align any two pairs an AP aligner must align all pairs while likely having overheads an M1:1 aligner would not, these overheads would make the AP aligner undesirable outside of its use case.

8. Boilerplate

Many of the implementations we consider follow a standard pattern:

1. Parse command-line arguments
2. Read data
3. Pack data onto GPU, if one is being used
4. Apply alignment kernels in parallel streams
5. Unpack results from GPU, if one is being used
6. Print performance metrics

For most implementations, the only part of the above that is unique and worth spending time on are the kernels. Every other piece is boilerplate; that is, code which is essentially the same from one implementation to the next. Furthermore, this boilerplate was often tightly integrated with other parts of the implementations making it difficult to apply unit tests to or refactor any part of the code.

In response, we have developed a library Alignment Boilerplate (ALBP, https://github.com/r-barnes/alignment_boilerplate) which abstracts this code into a reusable, unit-tested API. We describe this library in greater detail below. When we used this library to simplify GASAL2 [1] it reduced lines of code from 3,230 to 2,357 (27% reduction); using it on GPU-BSW [92] reduced lines of code from 1,608 to 468 (71% reduction). Performance was not reduced as a result.

The library provides abstractions for each of the steps in the pattern above:

- **Command-line Argument Parser.** As noted in § 7.2, a common cause of segfaults was a failure to check command-line arguments. Relatedly, a common cause of code bloat was parsing command-line arguments. ALBP solves both these problems by exposing the CLI11 command-line parser [76], which handles the full range of command-line functionality and provides easy methods for validating input.
- **FASTA handling.** ALBP provides functions for reading and interpreting FASTA files. In addition, it includes functions for packing FASTA data into the compressed forms needed by the GPU by producing start-index, end-index, and sequence-length arrays.
- **Invertible Sorts.** Sorting data by length to avoid hot-thread tail effects is an optimization strategy employed by several of the implementations we review. If a structure-of-arrays (SoA) data layout is used, the same sort may need to be applied to several arrays. Accordingly, ALBP includes functionality to determine the sorted positions of input data. These positions can then be used to accelerate subsequent sorts. They can also be used to invert the sorting when returning results to a user.
- **Ranges.** Many of the algorithms we review consider their inputs in batches. These batches, consisting of potentially thousands of sequences, are farmed out to GPU streams or CPU threads. Creating evenly-sized batches and handling leftovers leads to both code bloat and the potential for off-by-one errors. ALBP handles this by providing functions to create and manage ranges of indices corresponding to the lower and upper ends of batches.
- **Memory.** Writing safe CUDA code requires checking each API call for errors, ensuring that data types are propagated safely despite the CUDA API's extensive use of the `void` type, and freeing memory at appropriate times. ALBP simplifies this by using smart pointers (which automatically free memory) and templated, error-checking constructors. In contrast to the CUDA Thrust library, pointers are still the primary data structure. In addition,
 - ALBP couples ranges and memory to provide a safe, succinct way of moving memory to and from the GPU. ALBP also provides ways of handling page-locked memory.
- **Timer.** ALBP provides a Timer class for measuring the performance of code.
- **Stream Manager.** After FASTA data has been read it needs to be transferred to the GPU, used for computation, and results need to be transferred back. This occurs in batches and each batch is typically delegated to a separate stream. ALBP abstracts this process. A simple load balancer creates manager-threads for each GPU stream and/or CPU work-thread. The streams and work-threads themselves are managed by C++ functors which can hold and manage state specific to the stream. The Stream Manager assigns work in the form of Ranges (see above) to the manager-threads in round-robin fashion and the manager-threads pass this work off to their respective functors. Since the manager threads act only as delegators, over oversubscribing the CPU in this fashion does not affect performance while abstracting the need for the user to explicitly manage CPU threads to handle GPU streams. The functors themselves do not require an inheritance pattern since the C++ `functional` library allows functors to be stored based only on their input and return type. In the case, the input is a Range of data to be processed and no return is necessary: it is assumed that returns are written to memory and each functor has sole ownership of the region of memory referenced by the Range.
- **Unit Tests.** ALBP exposes the doctest [35] unit testing framework in addition to providing examples of the framework in use through its own unit tests.
- **CUDA Error Handling.** CUDA error handling often takes place through complex macro functions. ALBP simplifies the error handling process through a minimal set of macros that delegate quickly back to C++ code.
- **Simple Smith Waterman.** ALBP provides a naïve implementation of the SW algorithm. The correctness of this implementation is unit tested and is easily verified via visual inspection. A series of accompanying functions pro-

vide ways to slice and dice the naïve implementation’s output matrices. These tools make it easier to quickly write unit tests for new SW implementations.

ALBP also provides a couple of convenience programs:

- **FASTA Analyzer.** Extract sequence statistics from a FASTA file.
- **Random Sequence Generator.** Generates random datasets; useful for performance testing.

It is tempting to eliminate abstraction barriers to pursue full-stack optimization in search of additional performance, but SW is a poor candidate for this. The preparatory work the CPU does is always $O(N)$ (or $O(N \log N)$ if a comparison-based sort is used) in the number of sequences considered while the alignment itself takes approximately $O(Nmn)$ time in the number of sequences N and their lengths m and n . Since this is orders-of-magnitude more work, accelerating the alignments is a much better target for optimization effort. ALBP abstracts non-kernel functionality so that time can be spent on this.

9. Performance Comparison

Here, we compare the performance of several of the implementations listed in Table 2. Our performance comparison differs from previous ones in the literature in several respects:

- **Many implementations.** We compare performance across a large number of implementations. Most previous comparisons have either used authors’ self-reported performance or compared against only one or two other implementations.
- **Many input sizes.** We compare performance across a large range of input data sizes. Since GPU shared memory sizes and CPU cache sizes are limited, algorithms which work well for one length of data may perform much worse at other lengths. Most previous comparisons have focused on sequences that fall either within a narrow length range or have binned short and long sequences together making it impossible to separate their performance in reported numbers.

- **Reproducible.** None of the works reviewed include sufficient resources to reproduce their performance comparison tests. In contrast, we include an automated performance test rig and appropriately-modified source code for the implementations we test.

Since the Smith-Waterman algorithm fills in the cells of a dynamic programming matrix, we measure speed in billions of cells filled per second: GCUPS. We have modified the implementations’ code in our repository (https://github.com/r-barnes/sw_comparison) to output timing information in a standard format. Where possible, the timing excludes the time to read in the data. We have also built a benchmarking option which disables most file and command-line output, since this may vary considerably between programs. Under some conditions the CUDA compiler generates an intermediate representation which is converted to architecture-specific code the first time a program is run. We ensure this conversion doesn’t affect timing results by doing a warm-up run before each timing run. For GPU-only implementations we permit the use of 8 stream-management threads. For CPU-only implementations we use one thread. For mixed implementations, we explore GPU-only and CPU-only options when available. As a result, the performance values reflect GCUPS/GPU and GCUPS/thread.

We run our tests on a machine with an Intel Xeon CPU (2.20 GHz, 2 sockets, 4 cores/socket, 2 threads/core), 52 GB RAM, a 500 GB SSD hard drive, and 1 Nvidia V100 GPU.

To perform our tests we explore how different combinations of the lengths of the query and reference sequences affect performance. We adaptively vary the number of sequences tested to target a 15–20s test time by doubling the number of sequences in the test data until the test time falls beyond this band. This target test time is long enough to average out short-term speed variations within the test, but still short enough to be able to run all the tests in a reasonable amount of time. This also ensures that the GPU algorithms don’t experience data starvation.

Results of the performance comparisons are shown in Figure 1 and Figure 2. The black squares indicate benchmarks which failed because the underlying software crashed. We apply the black square if the crash happens for any amount of data considered, so it is possible that some of the software

would work for smaller datasets, even though the performance is likely to be lower due to data starvation.

Perhaps the most important result is that the CPU-based SSW [96] out-performs all of the other CPU-based methods we were able to test. This is surprising, given that it was released in 2013 and uses the SSE2 instruction set. New sets with additional primitives and wider vectors, such as SSE4 and AVX, have since become available. This suggests that additional research into CPU methods is likely warranted. SWIMM 2.0 [65] uses the AVX instruction set and claims to compare favorably against CUDASW++ 2.0 and 3.0, but we have not yet been able to verify these results because it only compiles with the (non-free) Intel compiler.

A second result is that SSW compares favorably against the GPU implementations. The results shown in Figure 2 are on a per-CPU-thread and per-GPU basis. Assuming, linear scaling on a 16-thread CPU, SSW’s single-thread performance of 2–7.1 GCUPS translates to 32–113.6 GCUPS. Even with sub-linear scaling, this still gives performance on-par with a single GPU. Thus, GPUs will be most beneficial for sequencing (a) if there are multiple of them, (b) the CPU can be engaged in other tasks, (c) they can be used in conjunction with the CPU, or (d) they can be used to save energy. If such conditions don’t hold, it likely makes more sense to use the CPU to handle sequencing tasks.

10. Future Work

Our review reveals several directions for future work.

Foremost is the need for aligners built with better software engineering. Of the implementations we reviewed, very few were designed for integration into other software; very few had unit tests; very few ran without crashing or raising errors interpretable to non-experts. All of these issues limit these implementations’ impact and usefulness. In addition to these basics, amenities such as wall-time estimates (§ 6) would be useful.

Of the implementations we review, the majority are built around an affine gap penalty while the rest are built around the special case of a linear penalty. This raises a question of whether the affine gap is so popular because it is the most biologically relevant

choice, or because the availability of increasingly performant algorithms using it creates a feedback cycle where no other option can be easily explored. The double-affine, convex, and general gap penalties are all open for exploration.

The M:1 motif has been explored by a number of groups. In contrast, the long S1:1 motif has been explored only the CUDAlign/MASA family and SW#. Additional attention may uncover better performance. The M1:1 motif is new, having only been explored by GASAL2 and GPU-BSW, so work here may also be fruitful. The AP motif is entirely unexplored. Beyond this, the ideas raised in § 5.1 regarding combining motifs to improved performance are unexplored. Relatedly, Table 2 and Figures 1 and 2 show that most aligners make strong assumptions about the number and lengths of their inputs. Were aligners designed as well-tested libraries, it would be possible to begin to combine them into meta-aligners which adaptively choose the algorithm and motif best suited to the input data.

While most performance metrics and comparisons to date, including ours, focus on GCUPS, instruction/integer roofline models are becoming available in GPUs [13] and may see adoption for easy use in profiling software. These profiling techniques offer an attractive way of determining which parts of an aligner are under-performing versus the theoretical capabilities of the GPU. In the future, wall-time may not even be the most important metric: the energy efficiency of algorithms is also becoming a concern (e.g. [59]).

Our performance comparison also reveals that a circa-2013 CPU implementation is competitive with the GPU implementations. This suggests that additional research on CPU implementations leveraging new instruction sets is likely worthwhile.

Finally, looking beyond GPUs, the future is full of exotic and apparently special-purpose hardware waiting to be leveraged, including FPGAs, APUs, quantum computers, nanophotonic accelerators, and more [18, 37, 52, 58, 64, 64, 66, 80, 97].

11. List of abbreviations

AA: Amino Acid, 20 of them comprise the building blocks of proteins, a kind of biological hardware; **ALBP:** Alignment Boilerplate (library); **CPU:**

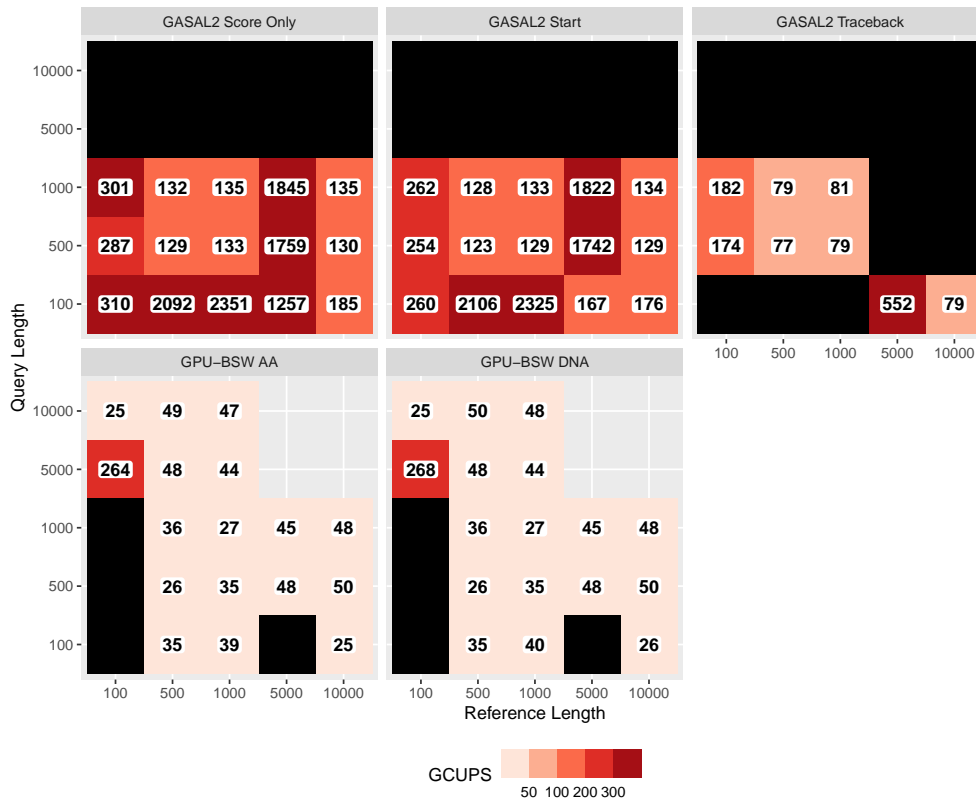


Figure 1: M1:1 performance comparison. Redder is better; bigger numbers are better. Numbers indicate GCUPS. Black squares indicate runs which crashed for some reason (see § 9 and § 15). Grey squares indicate test combinations which were not performed; GPU-BSW, for instance, has length limits on its inputs.

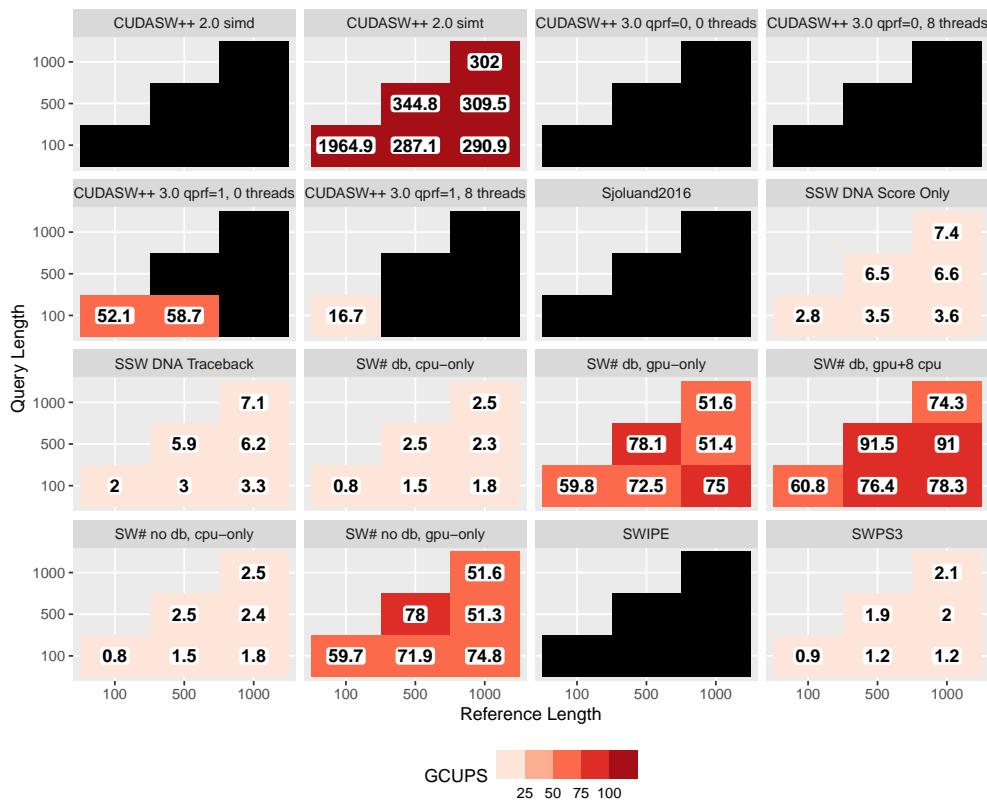


Figure 2: M:1 performance comparison. Redder is better; bigger numbers are better. Numbers indicate GCUPS. Black squares indicate runs which crashed for some reason (see § 9 and § 15). Grey squares indicate test combinations which were not performed; for instance, tests are constrained such that query lengths are always equal to or shorter than reference lengths.

Central Processing Unit; **CUDA**: Compute Unified Device Architecture (programming language); **DNA**: A fault-tolerant biological data storage format represented digitally with the characters A, C, G, T; **DP**: Dynamic Programming, the fine art of filling in matrices; **FASTA**: A file format used for storing DNA and RNA data; **FPGA**: Field-Programmable Gate Array; **GCUPS**: Billion Cell Updates per Second; **GPU**: Graphics Processing Unit; **OpenGL**: Open Graphics Library; **RAM**: Random Access Memory; **RNA**: A fault-prone biological data storage format used to encode proteins; **SIMD**: Single Instruction Multiple Data; **SIMT**: Single-instruction, Multiple-Thread; **SM**: Streaming Multiprocessor; **SSD**: Solid-State Hard-Drive; **SSE2**: Streaming SIMD Extensions 2; **SW**: Smith-Waterman

12. Software Availability

We have compiled the source code for 22 of the papers discussed here into a single repository. Many of the repositories include minor to major changes necessary to get the code to compile on a modern (Ubuntu 20.04) operating system. Each repository has been modified to use the `cmake` build system and benchmarking code has been added. The repository is available at: https://github.com/r-barnes/sw_comparison. In addition, we have developed Alignment Boilerplate (https://github.com/r-barnes/alignment_boilerplate), a well-documented, unit-tested package to reduce boilerplate in alignment software.

13. Acknowledgments

RB was supported by the Department of Energy’s Computational Science Graduate Fellowship (Grant No. DE-FG02-97ER25308) and, through the Berkeley Institute for Data Science’s PhD Fellowship, by the Gordon and Betty Moore Foundation (Grant GBMF3834) and by the Alfred P. Sloan Foundation (Grant 2013-10-27).

Empirical tests were performed on the Titan and Summitdev supercomputers managed by Oak Ridge National Laboratory’s Leadership Computing Facility; the Cori supercomputer managed by the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231; and XSEDE’s

Comet supercomputer [86], which is supported by the National Science Foundation (Grant No. ACI-1053575) and was available to RB through the NSF Graduate Research Fellowship.

Katherine Yelick and Aydın Buluç provided valuable input on the development of this project, as did Marquita Ellis and Muuaz Awan. Fernanda Foertter helped us upstream our modifications to NVBIO; Miheer Vaidya helped with these modifications. Kelly Kochanski helped proof the work.

14. Bibliography

References

- [1] Ahmed, N., Lévy, J., Ren, S., Mushtaq, H., Bertels, K., Al-Ars, Z., Dec. 2019. GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data. *BMC Bioinformatics* 20 (1), 520. doi: 10.1186/s12859-019-3086-9
- [2] Akoglu, A., Striemer, G. M., Sep. 2009. Scalable and highly parallel implementation of Smith-Waterman on graphics processing unit using CUDA. *Cluster Computing* 12 (3), 341–352. doi: 10.1007/s10586-009-0089-8
- [3] Aluru, S., Futamura, N., Mehrotra, K., 2003. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing* 63 (3), 264 – 272. doi: 10.1016/S0743-7315(03)00010-8
- [4] Azad, A., Pavlopoulos, G. A., Ouzounis, C. A., Kyrpides, N. C., Buluç, A., 01 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46 (6), e33–e33. doi: 10.1093/nar/gkx1313
- [5] Benner, S. A., Cohen, M. A., Gonnet, G. H., 1993. Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *Journal of Molecular Biology* 229 (4), 1065 – 1082. doi: 10.1006/jmbi.1993.1105
- [6] Burrows, M., Wheeler, D. J., 1994. A block-sorting lossless data compression algorithm. Tech. rep.
- [7] Bustamam, A., Ardaneswari, G., Lestari, D., Sep. 2013. Implementation of CUDA GPU-based parallel computing on Smith-Waterman algorithm to sequence database searches. In: 2013 International Conference on Advanced Computer Science and Information Systems (ICACSIS). IEEE, Sanur Bali, Indonesia, pp. 137–142. doi: 10.1109/ICACSIS.2013.6761565
- [8] Cartwright, R. A., Dec. 2006. Logarithmic gap costs decrease alignment accuracy. *BMC Bioinformatics* 7 (1), 527. doi: 10.1186/1471-2105-7-527
- [9] Czuma, A., 1996. Very fast approximation of the matrix chain product problem. *Journal of Algorithms* 21 (1), 71 – 79. doi: 10.1006/jagm.1996.0037
- [10] David, M., Dzamba, M., Lister, D., Ilie, L., Brudno, M., Apr. 2011. SHRiMP2: Sensitive yet practical short read mapping. *Bioinformatics* 27 (7), 1011–1012. doi: 10.1093/bioinformatics/btr046
- [11] Dayhoff, M., Schwartz, R., Orcutt, B., 1978. Chapter 22: A model of evolutionary change in proteins.

- In: Atlas of Protein Sequence and Structure. National Biomedical Research Foundation, pp. 345–352.
- [12] Dicker, A., Sibandze, B., Kelly, C., Mache, D., 2014. Viability of the parallel prefix algorithm for sequence alignment on massively parallel GPUs. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 1–4. URL <https://search.proquest.com/docview/1649687963>
- [13] Ding, N., Williams, S., 2019. An instruction roofline model for gpus. In: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 7–18.
- [14] Döring, A., Weese, D., Rausch, T., Reinert, K., Jan 2008. Seqan an efficient, generic c++ library for sequence analysis. BMC Bioinformatics 9 (1), 11. doi: 10.1186/1471-2105-9-11
- [15] Eddy, S. R., Aug 2004. Where did the BLOSUM62 alignment score matrix come from? Nature Biotechnology 22 (8), 1035–1036. doi: 10.1038/nbt0804-1035
- [16] Ellis, M., Guidi, G., Buluç, A., Olikier, L., Yelick, K., 2019. DiBELLA: Distributed long read to long read alignment. In: Proceedings of the 48th International Conference on Parallel Processing. ICPP 2019. Association for Computing Machinery, New York, NY, USA. doi: 10.1145/3337821.3337919
- [17] Farrar, M., Jan. 2007. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23 (2), 156–161. doi: 10.1093/bioinformatics/btl582
- [18] Fei, X., Dan, Z., Lina, L., Xin, M., Chunlei, Z., Mar. 2018. FPGASW: Accelerating Large-Scale Smith–Waterman Sequence Alignment Application with Backtracking on FPGA Linear Systolic Array. Interdisciplinary Sciences: Computational Life Sciences 10 (1), 176–188. doi: 10.1007/s12539-017-0225-8
- [19] Galil, Z., Giancarlo, R., 1989. Speeding up dynamic programming with applications to molecular biology. Theoretical Computer Science 64 (1), 107 – 118. doi: 10.1016/0304-3975(89)90101-1
- [20] Gotoh, O., 1982. An improved algorithm for matching biological sequences. Journal of Molecular Biology 162 (3), 705 – 708. doi: 10.1016/0022-2836(82)90398-9
- [21] Gotoh, O., Mar. 2000. Homology-based gene structure prediction: simplified matching algorithm using a translated codon (tron) and improved accuracy by allowing for long gaps. Bioinformatics 16 (3), 190–202. doi: 10.1093/bioinformatics/16.3.190
- [22] Gu, X., Li, W.-H., Apr. 1995. The size distribution of insertions and deletions in human and rodent pseudogenes suggests the logarithmic gap penalty for sequence alignment. Journal of Molecular Evolution 40 (4), 464–473. doi: 10.1007/BF00164032
- [23] Guidi, G., Ellis, M., Rokhsar, D., Yelick, K., Buluç, A., 2020. BELLA: Berkeley efficient long-read to long-read aligner and overlapper. bioRxiv. doi: 10.1101/464420
- [24] Gupta, P., 2012. Swift: A GPU-based Smith-Waterman sequence alignment program.
- [25] Hains, D., Cashero, Z., Ottenberg, M., Bohm, W., Rajopadhye, S., May 2011. Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. IEEE, Anchorage, AK, USA, pp. 490–501. doi: 10.1109/IPDPS.2011.182
- [26] He, S., Malfatti, S. A., McFarland, J. W., Anderson, F. E., Pati, A., Huntemann, M., Tremblay, J., Glavina del Rio, T., Waldrop, M. P., Windham-Myers, L., Tringe, S. G., 2015. Patterns in wetland microbial community composition and functional gene repertoire associated with methane emissions. mBio 6 (3). doi: 10.1128/mBio.00066-15
- [27] Henikoff, S., Henikoff, J. G., Nov 1992. Amino acid substitution matrices from protein blocks. Proceedings of the National Academy of Sciences of the United States of America 89 (22), 10915–10919. doi: 10.1073/pnas.89.22.10915
- [28] Hess, M., Keul, F., Goesele, M., Hamacher, K., Apr 2016. Addressing inaccuracies in BLOSUM computation improves homology search performance. BMC Bioinformatics 17, 189–189. doi: 10.1186/s12859-016-1060-3
- [29] Hu, T. C., Shing, M. T., 1982. Computation of matrix chain products. Part I. SIAM Journal on Computing 11 (2), 362–373. doi: 10.1137/0211028
- [30] Huang, L.-T., Wu, C.-C., Lai, L.-F., Li, Y.-J., 2015. Improving the Mapping of Smith-Waterman Sequence Database Searches onto CUDA-Enabled GPUs. BioMed Research International 2015, 1–10. doi: 10.1155/2015/185179
- [31] Jia, Z., Maggioni, M., Smith, J., Scarpazza, D. P., 2019. Dissecting the Nvidia Turing T4 GPU via microbenchmarking. arXiv:1903.07486.
- [32] Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D. P., 2018. Dissecting the Nvidia Volta GPU architecture via microbenchmarking. arXiv: 1804.06826.
- [33] Jones, D. T., Taylor, W. R., Thornton, J. M., 06 1992. The rapid generation of mutation data matrices from protein sequences. Bioinformatics 8 (3), 275–282. doi: 10.1093/bioinformatics/8.3.275
- [34] Khajeh-Saeed, A., Poole, S., Blair Perot, J., Jun. 2010. Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors. Journal of Computational Physics 229 (11), 4247–4258. doi: 10.1016/j.jcp.2010.02.009
- [35] Kirilov, V., 2020. doctest: C++ testing framework. <https://github.com/onqtam/doctest>, v2.4.0, 1c8da00.
- [36] Klus, P., Lam, S., Lyberg, D., Cheung, M., Pullan, G., McFarlane, I., Yeo, G. S., Lam, B. Y., 2012. BaraCUDA - a fast short read sequence aligner using graphics processing units. BMC Research Notes 5 (1), 27. doi: 10.1186/1756-0500-5-27
- [37] Krommydas, K., Feng, W.-c., Antonopoulos, C. D., Bellas, N., Dec. 2016. OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures. Journal of Signal Processing Systems 85 (3), 373–392. doi: 10.1007/s11265-015-1051-z
- [38] Langmead, B., Salzberg, S. L., 2012. Fast gapped-read alignment with Bowtie 2. Nature Methods 9 (4), 357–359. doi: 10.1038/nmeth.1923
- [39] Ligowski, L., Rudnicki, W., May 2009. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE, Rome, Italy, pp. 1–8. doi: 10.1109/IPDPS.2009.5160931
- [40] Ling, C., Benkrid, K., Hamada, T., Jul. 2009. A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. In: 2009

- IEEE 7th Symposium on Application Specific Processors. IEEE, San Francisco, CA, pp. 94–100. doi: 10.1109/SASP.2009.5226343
- [41] Liu, Y., Huang, W., Johnson, J., Vaidya, S., 2006. GPU Accelerated Smith-Waterman. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Alexandrov, V. N., van Albada, G. D., Sloat, P. M. A., Dongarra, J. (Eds.), Computational Science – ICCS 2006. Vol. 3994. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 188–195. doi: 10.1007/11758549_29
- [42] Liu, Y., Maskell, D. L., Schmidt, B., 2009. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Research Notes 2 (1), 73. doi: 10.1186/1756-0500-2-73
- [43] Liu, Y., Schmidt, B., Mar. 2015. GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences: GSWABE: faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. Concurrency and Computation: Practice and Experience 27 (4), 958–972. doi: 10.1002/cpe.3371
- [44] Liu, Y., Schmidt, B., Maskell, D. L., Dec. 2010. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMD and virtualized SIMD abstractions. BMC Research Notes 3 (1), 93. doi: 10.1186/1756-0500-3-93
- [45] Liu, Y., Wirawan, A., Schmidt, B., Dec. 2013. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinformatics 14 (1), 117. doi: 10.1186/1471-2105-14-117
- [46] Luo, R., Wong, T., Zhu, J., Liu, C.-M., Zhu, X., Wu, E., Lee, L.-K., Lin, H., Zhu, W., Cheung, D. W., Ting, H.-F., Yiu, S.-M., Peng, S., Yu, C., Li, Y., Li, R., Lam, T.-W., May 2013. SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner. PLoS ONE 8 (5), e65632. doi: 10.1371/journal.pone.0065632
- [47] Manavski, S. A., Valle, G., Mar. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9 (S2), S10. doi: 10.1186/1471-2105-9-S2-S10
- [48] Marcos, N., 2014. Efficient GPU implementation of bioinformatics applications. Master’s thesis, Instituto Superior Técnico, Department of Information Systems and Computer Engineering, Lisbon, Portugal.
- [49] Miller, W., Myers, E. W., Mar. 1988. Sequence comparison with concave weighting functions. Bulletin of Mathematical Biology 50 (2), 97–120. doi: 10.1007/BF02459948
- [50] Munekawa, Y., Ino, F., Hagihara, K., Oct. 2008. Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU. In: 2008 8th IEEE International Conference on Bioinformatics and BioEngineering. IEEE, Athens, Greece, pp. 1–6. doi: 10.1109/BIBE.2008.4696721
- [51] Myers, E. W., Miller, W., 03 1988. Optimal alignments in linear space. Bioinformatics 4 (1), 11–17. doi: 10.1093/bioinformatics/4.1.11
- [52] Müller, A., Schmidt, B., Hildebrandt, A., Membarth, R., Leiða, R., Kruse, M., Hack, S., Feb. 2020. AnySeq: A high performance sequence alignment library based on partial evaluation. arXiv:2002.04561 [cs]ArXiv: 2002.04561.
- [53] Okada, D., Ino, F., Hagihara, K., Dec. 2015. Accelerating the Smith-Waterman algorithm with interpair pruning and band optimization for the all-pairs comparison of base sequences. BMC Bioinformatics 16 (1), 321. doi: 10.1186/s12859-015-0744-4
- [54] Okonechnikov, K., Golosova, O., Fursov, M., The UGENE Team, Feb. 2012. Unipro UGENE: a unified bioinformatics toolkit. Bioinformatics 28 (8), 1166–1167. doi: 10.1093/bioinformatics/bts091
- [55] Pantaleoni, J., Subtil, N., Barnes, R., Vaidya, M., Chen-Zhihui, Foertter, F., Rehfeld, H., 2015. Nvbio. <https://github.com/NVlabs/nvbio>.
- [56] Peltonen, L., McKusick, V. A., 2001. Dissecting human disease in the postgenomic era. Science 291 (5507), 1224–1229. doi: 10.1126/science.291.5507.1224
- [57] Prasad, D. V. V., Jaganathan, S., Jul. 2019. Improving the performance of Smith-Waterman sequence algorithm on GPU using shared memory for biological protein sequences. Cluster Computing 22 (S4), 9495–9504. doi: 10.1007/s10586-018-2421-7
- [58] Prousalis, K., Konofaos, N., 2017. Quantum pattern recognition for local sequence alignment. In: 2017 IEEE Globecom Workshops (GC Wkshps). IEEE, pp. 1–5.
- [59] Pérez-Serrano, J., Sandes, E. F., Magalhaes Alves de Melo, A. C., Ujaldón, M., Nov. 2018. DNA sequences alignment in multi-GPUs: acceleration and energy payoff. BMC Bioinformatics 19 (S14), 421. doi: 10.1186/s12859-018-2389-6
- [60] Qin, J., Li, R., Raes, J., Arumugam, M., Burgdorf, K. S., Manichanh, C., Nielsen, T., Pons, N., Levenez, F., Yamada, T., Mende, D. R., Li, J., Xu, J., Li, S., Li, D., Cao, J., Wang, B., Liang, H., Zhang, H., Xie, Y., Tap, J., Lepage, P., Bertalan, M., Batto, J.-M., Hansen, T., Le Paslier, D., Linneberg, A., Nielsen, H. B., Pelletier, E., Renault, P., Sicheritz-Ponten, T., Turner, K., Zhu, H., Yu, C., Li, S., Jian, M., Zhou, Y., Li, Y., Zhang, X., Li, S., Qin, N., Yang, H., Wang, J., Brunak, S., Doré, J., Guarner, F., Kristiansen, K., Pedersen, O., Parkhill, J., Weissenbach, J., Antolin, M., Artiguenave, F., Blottiere, H., Borruel, N., Bruls, T., Casellas, F., Chervaux, C., Cultrone, A., Delorme, C., Denariac, G., Dervyn, R., Forte, M., Friss, C., van de Guchte, M., Guedon, E., Haimet, F., Jamet, A., Juste, C., Kaci, G., Kleerebezem, M., Knol, J., Kristensen, M., Layec, S., Le Roux, K., Leclerc, M., Maguin, E., Melo Minardi, R., Oozeer, R., Rescigno, M., Sanchez, N., Tims, S., Torrejon, T., Varela, E., de Vos, W., Winogradsky, Y., Zoetendal, E., Bork, P., Ehrlich, S. D., Wang, J., Consortium, M., Mar 2010. A human gut microbial gene catalogue established by metagenomic sequencing. Nature 464 (7285), 59–65. doi: 10.1038/nature08821
- [61] Rice Genome Sequencing Project International, 2005. The map-based sequence of the rice genome. Nature 436 (7052), 793.
- [62] Rognes, T., Dec. 2011. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. BMC Bioinformatics 12 (1), 221. doi: 10.1186/1471-2105-12-221
- [63] Rucci, E., De Giusti, A., Naiouf, M., Botella, G., Garcia, C., Prieto-Matias, M., Sep. 2014. Smith-Waterman algorithm on heterogeneous systems: A case study. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, Madrid, Spain, pp. 323–

330. doi: 10.1109/CLUSTER.2014.6968784
- [64] Rucci, E., Garcia, C., Botella, G., De Giusti, A., Naiouf, M., Prieto-Matias, M., Nov. 2018. SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences. *BMC Systems Biology* 12 (S5), 96. doi: 10.1186/s12918-018-0614-6
- [65] Rucci, E., Garcia Sanchez, C., Botella Juan, G., Giusti, A. D., Naiouf, M., Prieto-Matias, M., Apr. 2018. SWIMM 2.0: Enhanced Smith-Waterman on Intel's Multicore and Manycore Architectures Based on AVX-512 Vector Extensions. *International Journal of Parallel Programming* 47 (2), 296–316. doi: 10.1007/s10766-018-0585-7
- [66] Rucci, E., García, C., Botella, G., De Giusti, A., Naiouf, M., Prieto-Matías, M., Dec. 2015. An energy-aware performance analysis of SWIMM: Smith-Waterman implementation on intel's multicore and manycore architectures. *Concurrency and Computation: Practice and Experience* 27 (18), 5517–5537. doi: 10.1002/cpe.3598
- [67] Rumble, S. M., Lacroute, P., Dalca, A. V., Fiume, M., Sidow, A., Brudno, M., May 2009. SHRiMP: Accurate mapping of short color-space reads. *PLoS Computational Biology* 5 (5), e1000386. doi: 10.1371/journal.pcbi.1000386
- [68] Sandes, E. F., de Melo, A. C. M., May 2011. Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space. *IEEE*, pp. 1199–1211. doi: 10.1109/IPDPS.2011.114
- [69] Sandes, E. F., de Melo, A. C. M., May 2013. Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences Using GPU. *IEEE Transactions on Parallel and Distributed Systems* 24 (5), 1009–1021. doi: 10.1109/TPDS.2012.194
- [70] Sandes, E. F., Melo, A. C. M. d., 2010. CUDAAlign: using GPU to accelerate the comparison of megabase genomic sequences. p. 10. doi: 10.1145/1693453.1693473
- [71] Sandes, E. F., Miranda, G., Martorell, X., Ayguade, E., Teodoro, G., De Melo, A. C. M. A., Feb. 2016. MASA: A Multiplatform Architecture for Sequence Aligners with Block Pruning. *ACM Transactions on Parallel Computing* 2 (4), 1–31. doi: 10.1145/2858656
- [72] Sandes, E. F., Miranda, G., Martorell, X., Ayguade, E., Teodoro, G., Melo, A. C. M., Oct. 2016. CUDAAlign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 27 (10), 2838–2850. doi: 10.1109/TPDS.2016.2515597
- [73] Sandes, E. F., Miranda, G., Melo, A. C., Martorell, X., Ayguade, E., 2014. Fine-grain parallel megabase sequence comparison with multiple heterogeneous GPUs. *ACM Press*, pp. 383–384. doi: 10.1145/2555243.2555280
- [74] Sandes, E. F., Miranda, G., Melo, A. C. d., Martorell, X., Ayguade, E., May 2014. CUDAAlign 3.0: Parallel biological sequence comparison in large GPU clusters. *IEEE*, pp. 160–169. doi: 10.1109/CCGrid.2014.18
- [75] Sandes, E. F., Teodoro, G. L. M., Walter, M. E. M. T., Martorell, X., Ayguade, E., Melo, A. C. M. A., May 2018. Formalization of block pruning: Reducing the number of cells computed in exact biological sequence comparison algorithms. *The Computer Journal* 61 (5), 687–713. doi: 10.1093/comjnl/bxx090
- [76] Schreiner, H., 2020. CLI11: Command line parser for C++11. <https://github.com/CLIUtils/CLI11>, v1.9.1, 5cb3efa.
- [77] Sjölund, E., Lindahl, E., 2016. diagonalsw. <https://github.com/eriksjoelund/diagonalsw>.
- [78] Smith, T., Waterman, M., Mar. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147 (1), 195–197. doi: 10.1016/0022-2836(81)90087-5
- [79] States, D. J., Gish, W., Altschul, S. F., 1991. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *Methods* 3 (1), 66 – 70. doi: 10.1016/S1046-2023(05)80165-3
- [80] Steinfadt, S., Dec. 2013. Fine-grained parallel implementations for SWAMP+ Smith-Waterman alignment. *Parallel Computing* 39 (12), 819–833. doi: 10.1016/j.parco.2013.08.008
- [81] Striemer, G., Akoglu, A., May 2009. Sequence alignment with GPU: Performance and design challenges. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. *IEEE*, Rome, pp. 1–10. doi: 10.1109/IPDPS.2009.5161066
- [82] Styczynski, M. P., Jensen, K. L., Rigoutsos, I., Stephanopoulos, G., Mar 2008. BLOSUM62 miscalculations improve search performance. *Nature Biotechnology* 26 (3), 274–275. doi: 10.1038/nbt0308-274
- [83] Sung, W.-K., 2009. Algorithms in Bioinformatics: A Practical Introduction. Chapman and Hall/CRC.
- [84] Szalkowski, A., Ledergerber, C., Krähenbühl, P., Dessimoz, C., 2008. SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes* 1 (1), 107. doi: 10.1186/1756-0500-1-107
- [85] Tatusova, T. A., Madden, T. L., 1999. BLAST 2 sequences, a new tool for comparing protein and nucleotide sequences. *FEMS microbiology letters* 174 (2), 247–250.
- [86] Towns, J., Cockerill, T., Dahan, M., Foster, I., Gathier, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G. D., Roskies, R., Scott, J. R., Wilkins-Diehr, N., Sep. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering* 16 (5), 62–74. doi: 10.1109/MCSE.2014.80
- [87] Warris, S., Timal, N. R. N., Kempenaar, M., Poortinga, A. M., van de Geest, H., Varbanescu, A. L., Nap, J.-P., Jan. 2018. pyPaSWAS: Python-based multi-core CPU and GPU sequence alignment. *PLOS ONE* 13 (1), e0190279. doi: 10.1371/journal.pone.0190279
- [88] Warris, S., Yalcin, F., Jackson, K. J. L., Nap, J. P., Apr. 2015. Flexible, Fast and Accurate Sequence Alignment Profiling on GPGPU with PaSWAS. *PLOS ONE* 10 (4), e0122524. doi: 10.1371/journal.pone.0122524
- [89] Waterman, M. S., Jun. 1984. Efficient sequence alignment algorithms. *Journal of Theoretical Biology* 108 (3), 333–337. doi: 10.1016/S0022-5193(84)80037-5
- [90] Williams, L., Kudrjavets, G., Nagappan, N., 2009. On the effectiveness of unit test automation at Microsoft. In: 2009 20th International Symposium on Software Reliability Engineering. pp. 81–89. doi: 10.1109/IS-SRE.2009.32
- [91] Wirawan, A., Kwok, C. K., Hieu, N. T., Schmidt, B., Sep 2008. CBESW: Sequence alignment on the playstation 3. *BMC Bioinformatics* 9 (1), 377. doi: 10.1186/1471-2105-9-377
- [92] Yelick, K. A., Oliker, L., Awan, M. G., 2020. GPU accelerated Smith-Waterman for performing batch alignments (GPU-BSW) v1.0. <https://www.osti.gov/biblio/1580217>. doi: 10.11578/dc.20191223.1

- [93] Yongchao Liu, Schmidt, B., Feb. 2014. CUSHAW2-GPU: Empowering faster gapped short-read alignment using GPU computing. *IEEE Design & Test* 31 (1), 31–39. doi: 10.1109/MDAT.2013.2284198
- [94] Zeni, A., Guidi, G., Ellis, M., Ding, N., Santambrogio, M. D., Hofmeyr, S., Buluç, A., Olikier, L., Yelick, K., 2020. LOGAN: High-performance GPU-based X-Drop long-read alignment. arXiv:2002.05200.
- [95] Zhang, Z., Schwartz, S., Wagner, L., Miller, W., 2000. A greedy algorithm for aligning dna sequences. *Journal of Computational Biology* 7 (1-2), 203–214, PMID: 10890397. doi: 10.1089/10665270050081478
- [96] Zhao, M., Lee, W.-P., Garrison, E. P., Marth, G. T., Dec. 2013. SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications. *PLOS ONE* 8 (12), e82138. doi: 10.1371/journal.pone.0082138
- [97] Zou, H., Tang, S., Yu, C., Fu, H., Li, Y., Tang, W., 2019. ASW: Accelerating Smith-Waterman algorithm on coupled CPU-GPU architecture. *International Journal of Parallel Programming* 47 (3), 388–402. doi: 10.1007/s10766-018-0617-3

Table 2: Aspects of CPU and GPU sequencing software

	Name	Year	Tech	Has Code?	Compiles?	Has Tests?	Library?	Known to Crash?	Penalty	Motif	Matrix	Traceback?	Lengths
CPU	F-2007* [17]	2007	SSE2	✗					affine	S1:1	BLS	✗	((1H-6H)v364)x208K
	★ SWPS3 [84]	2008	SSE2	✓	✓	✗	✓		affine	S1:1	BLS	✗	((1H-4K)v359)x360K
	SHRiMP [67]	2009	SIMD	✗					affine	M:1	MM	✗	(35v??)x135M
	SHRiMP2 [10]	2011	SIMD	✓	✓	✗	✗		affine	M:1	MM	✓	(75v??)x6M
	★ SWIPE [62]	2011	SSSE3	✓	✓	✗	✗		affine	M:1	BLS	✗	(375v326)x5M
	bowtie2 [38]	2012		✓	✓	✗	✗						
	★ SSW [96]	2013	SIMD	✓	✓	✗	✓		affine	S1:1	BLS	✓	(1Hv100M)x1K
	SWIMM [63]	2014	Xeon Phi	✗					affine	M:1	BLS	✗	((1H-5K)v355)x541K
	SWIMM [66]	2015	Xeon Phi	✓	✓	✗	✗		affine		BLS	✗	
	★ DiagonalSW [77]	2016	SSE4/Altivec	✓	✓	✗	✗		affine	M:1	BLS	✗	??
GPU	SWIMM2.0 [65]	2018	AVX512	✓	✗ ^{§15.15}	✗	✗		affine	M:1	BLS	✗	((1H-5K)v335)x710K
	LHJV-2006* [41]	2006	OpenGL	✗					double	M:1	BLS	✓	(16Kv471)x983
	MIH-2008* [50]	2008	CUDA	✗					linear ^{§15.2}	M:1	??	✗	((63-511)v362)x90M
	SWCUDA [47]	2008	CUDA	✓	✗	✗	✗		affine	M1:1	BLS	✗	((63-5H)v366)x250K
	Liu2009* [42]	2009	CUDA	✗					affine	M:1	BLS	✗	((144-59K)v??)x??
	AS-2009* [2]	2009	CUDA	✗					linear	M:1	BLS	✗	((4-1K)v??)x393K
	LR-2009* [39]	2009	CUDA	✗					affine	M:1	BLS	✗	((10-1K)v1K)x389K
	GSW [81]	2009	CUDA	✓	✓	✗	✗	✓ ^{§15.10}	linear	M:1	BLS	✗	((4-1K)v??)x393K
	LBH-2009* [40]	2009	CUDA	✗					linear	M:1	BLS	✗	((63-4K)v??)x400K
	CUDAalign1.0 [70]	2010	CUDA	✗					affine	S1:1	MM	✗	((2H-33M)v(2H-47M))x1
	★ CUDASW++ 2.0 [44]	2010	CUDA	✓	✓	✗	✗	✓ ^{§15.11}	affine	M:1	BLS	✗	((1H-5K)v??)x406K
	KPP-2010* [34]	2010	CUDA	✓	✓	✗	✗	✓ ^{§15.14}	affine	S1:1	MM	✓	1Kv(10M-10G)
	HBR-2011* [25]	2011	CUDA	✗					linear	M:1	BLS	✗	((10-5K)v??)x400K
	CUDAalign2.0 [68]	2011	CUDA	✗					affine	S1:1	MM	✓	((2H-33M)v(2H-47M))x1
	BarraCUDA [36]	2012	CUDA	✓	✓	✗	✗		affine	M:1	MM	✗	((37-67)v102M
	SWIFT [24]	2012	CUDA	✓	✓*	✓	✗	✓ ^{§15.9}	affine	??	MM	✗	(1Hv3B)x14M
	BAL-2013* [7]	2013	CUDA	✗					affine	M:1	BLS	✗	((1H-2K)v??)x400K
	CUDAalign2.1 [69]	2013	CUDA	✗					affine	S1:1	MM	✓	((162K-59M)v(162K-59M))x1
	CUSHAW2-GPU [93]	2014	CUDA	✓	✓	✗	✗		affine	M:1	MM	✓	((1H-2H)v3G)x1M
	MASA [73]	2014	CUDA	✗					affine	S1:1	MM	✓	((46M-64M)v(46M-64M))x1
	CUDAalign3.0 [74]	2014	CUDA	✗					affine	S1:1	MM	✓	(228M v 228M)x1
	PaSWAS [88]	2015	CUDA	✓	✓*	✗	✗		linear	M:1	BLS	✓	(5v360)x402K
	★ SW# [53]	2015	CUDA	✓	✓	✗	✓	✓ ^{§15.12}	affine	S1:1	MM	✗	((5M-64M)v(5M-64M))x1
	HWLL-2015* [30]	2015	CUDA	✗					affine	M:1	BLS	✗	??
	★ NVBIO [55]	2015	CUDA	✗ ^{§15.7}	✓	✓	✓	✓ ^{§15.7}	affine	M:1	BLS	✓	
	CUDAalign4.0 [72]	2016	CUDA	✗					affine	S1:1	MM	✓	(249M v 249M)x1
	MASA/CUDAalign [71]	2016	CUDA	✓	✓	✗	✓		affine	S1:1	MM	✓	
	★ GASAL2 [1]	2019	CUDA	✓	✓	✗	✓	✓ ^{§15.4}	affine	M1:1	MM	✓	((1H-3H)v(2H-6H))x10M
	PJ-2019* [57]	2019	CUDA	✗					affine ^{§15.3}	M:1	??	✓	((60-10K)v(3K-20K))x14K
	★ GPU-BSW [92]	2020	CUDA	✓	✓	✗	✗	✓ ^{§15.5}	affine	M1:1	BLS	✗	(1Kv1K)x30K
ugene [54]	2020	CUDA	✓	✓	✓	✗	✓ ^{§15.8}	affine	M:1	BLS	✓	??	

Table 2 (continued) — Aspects of CPU and GPU sequencing software

	Name	Year	Tech	Has Code?	Compiles?	Has Tests?	Library?	Known to Crash?	Penalty	Motif	Matrix	Traceback?	Lengths
CPU+GPU	AnySeq [52]	2020	AnyDSL	✓	✗ ^{§15.1}	✗	✗		affine	M/S 1:1	MM	✓	(50Mv50M) & (1Hv1H)x13M
	★ CUDASW++ 3.0 [45]	2013	SSE/CUDA	✓	✓	✗	✗	✓ ^{§15.6}	affine	M:M	BLS	✗	((1H-5K)v355)x539K
	SOAP3 [46]	2013		✓	✓	✗	✓		affine	M:1?			
	M-2014* [48]	2014		✗					??	??	??		??
	pyPaSWAS [87]	2018	Python	✓	✓	✓	✗		affine	M:M	BLS		??

Table 2: Implementation summary. Names marked with a star (★) are included in the performance comparison in § 9. Names labeled with asterisks (*) correspond to works without a specific name; in this case, names have been generated based on the first authors’ last names and the year of publication. Tech refers to the primary hardware or language features exploited by the implementation. Penalty refers to the gap penalties listed in § 2. Motif (§ 5) is either Many-to-One (M:1), Single One-to-One (S1:1), or Many One-to-One (M1:1). Matrix refers to the substitution matrix (§ 3) which is either BLOSUM (BLS) or Match-Mismatch (MM). Length denotes the lengths of the sequences compared. Suffices represent metric multipliers (H=Hundred, K=Thousand, M=Million, G=Billion). Alignments between sequences of different lengths are denoted AvB where A and B are lengths or length ranges. The number N of such alignments performed is denoted by xN. Averages are denoted with an overbar. “??” implies a value which could not be determined from either the publication or the associated code.

15. Appendix: Software Issues

15.1. AnySeq

Reference: Müller et al. [52]

Builds for standard CPU.

Fails to build for AVX unless LLVM is recompiled from source with RV support from <https://github.com/cdl-saarland/rv>. While doing so might resolve the issue, this is likely too much effort for most users to consider it feasible.

Fails to build for CUDA, raising the error message: `anyseq/src/traceback.impala:90 col 9 - 27: currently only pointers to arrays supported as kernel argument; argument has different type: qs32*`

15.2. MIH-2008

Reference: Munekawa et al. [50]

Uses a linear penalty gap hard-coded to 1.

15.3. PJ-2019

Reference: Prasad and Jaganathan [57]

This algorithm seems to use the SW without the E and F matrices, unacceptably increasing its computational complexity.

15.4. GASAL2

Reference: Ahmed et al. [1]

`b9adbb6` crashes with CUDA errors when performing a traceback; this appears to be a memory allocation issue. See <https://github.com/nahmedraj/GASAL2/issues/5>.

In both the score-only and find-start modes CUDA illegal memory errors were returned when trying to sequence (5Kv10K)x1M, (5Kv5K)x1M, (10Kv10K)x1M, (5Kv10K)x1K, (5Kv5K)x1K, (10Kv10K)x1K. During Traceback CUDA out-of-memory errors were returned for both 1K and 1M datasets of (100v5000), (100v10000), (500v1000), (500v5000), (500v10000), (1000v5000), (1000v10000), (5000v10000), (1000v1000), (5000v5000), (10000v10000).

15.5. GPU-BSW

Reference: Yelick et al. [92]

`cbe9ddd` doesn't describe its input format, leading to immediate segfaults upon running. Compilation targets a specific Nvidia Compute Capability resulting in silent kernel failures on GPUs with other compute capabilities due to unchecked errors. Differences between the AA and DNA kernel suggest a bug, though this does not manifest in an obvious way. Sequences that are too long are silently truncated. We fix these issues in a forked repository at <https://github.com/r-barnes/GPU-BSW>.

15.6. CUDASW++3.0

Reference: Liu et al. [45]

Running with `qprf=1` and no threads for (1Kx10)v(10Kx1) and (1Kx10)v(5Kx10) gave a CUDA invalid configuration argument error. Running with `qprf=0` and no threads for (1Kx10)v(10Kx1), (1Kx10)v(5Kx10), and (100x1K)v(100x200) gave a CUDA invalid configuration argument and an invalid argument error. For `qprf=1` and 8 threads (1Kx100)v(100x200) gave a `malloc invalid next size` error. A number of other sizes were similarly affected.

15.7. NVBIO

Reference: Pantaleoni et al. [55]

`9ea05dedd` only compiles with CUDA 6.5 and GCC 4.8.2, making this difficult to compile on many modern operating systems. We made a large number of changes (since upstreamed) to produce `5916f3ea`, which compiles with CUDA10 and GCC8. Unexpected inputs can lead to segfaults. Non-existent input files give segfaults.

15.8. UGENE

Reference: Okonechnikov et al. [54]

`bd87ca4` crashes when the UI is run, complaining about missing language files. The CLI seems to run fine.

15.9. SWIFT

Reference: Gupta [24]

Segfaults when run as `swift -m 1 -M -4 -O -6 -E -1 -o /dev/null -q QUERYFILE -r REFFILE` where QUERYFILE and REFFILE are FASTA files each of which contain 1,000 sequences of 300 bases.

15.10. GSW

Reference: Striemer and Akoglu [81]

Runs without crashing, but the query sequence is hard-coded.

15.11. CUDASW++ 2.0

Reference: Liu et al. [44]

Raises a CUDA invalid argument error in SIMT mode for (1Kx10)v(5Kx10) and (1Kx10)v(10Kx1). Every attempt to use SIMD mode results in a CUDA illegal argument error.

15.12. SW#

Reference: Okada et al. [53]

Trying to perform a GPU-only alignment between to 5M base sequences gives a CUDA invalid argument error.

15.13. SWIMM

Reference: Rucci et al. [66]

Requires a Xeon Phi, which we did not have available during testing.

15.14. KPP-2010

Reference: Khajeh-Saeed et al. [34]

The program takes no inputs and ran for a half hour before we manually terminated it.

15.15. SWIMM 2.0

Reference: Rucci et al. [65]

Requires the (non-free) Intel compiler, which was not available to us at the time of testing.