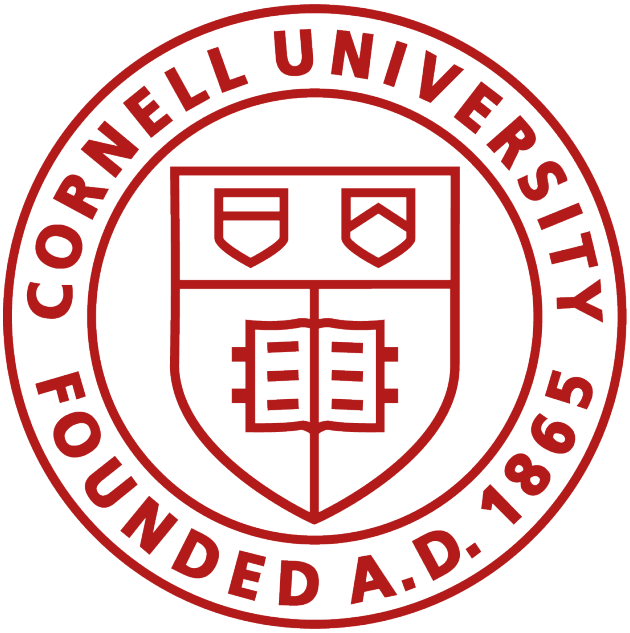
Accelerating Smith-Waterman with High-Level Synthesis

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering



Submitted by
Lawrence Atienza

MEng Advisor: Christopher Batten

Degree Date: May 2025

1 Abstract

Project Title: Accelerating Smith-Waterman with High Level Synthesis

Author: Lawrence Atienza

Abstract: This report explores the implementation of the Smith-Waterman algorithm—a fundamental local sequence alignment method in computational genomics—using Field-Programmable Gate Arrays (FPGAs) through High-Level Synthesis (HLS). Smith-Waterman identifies optimal matching subsequences between biological sequences, making it crucial for detecting conserved domains and evolutionary relationships. Despite its algorithmic power, Smith-Waterman's O(N^2) complexity, fine-grained data dependencies, and memory bandwidth limitations pose significant implementation challenges, particularly for large datasets.

FPGAs offer promising acceleration capabilities through custom parallelism and optimized memory access patterns, potentially outperforming CPUs and GPUs in both throughput and energy efficiency. While traditional FPGA development using hardware description languages enables fine-grained control, it substantially increases development complexity. HLS tools address this challenge by allowing hardware behavior specification in high-level languages like C/C++, significantly reducing development time.

This work presents a comprehensive design space exploration of Smith-Waterman FPGA implementations using HLS, comparing performance against software baselines. Results show that my optimized FPGA implementation achieves calculation times within 2× of off-the-shelf baselines and my own parallelized baseline, though with worse energy efficiency than the most efficient CPU designs. The study demonstrates HLS' capability to rapidly produce competitive designs, while highlighting limitations in resource control and optimization visibility for routing and timing constraints.

Contents

2 Introduction

Since the advent of advanced genome-editing technologies such as CRISPR in 2012, the volume of genetic data generated has grown exponentially. This surge in data has led to significantly increased computational demands for processing and analysis. Consequently, the development of novel, high-efficiency algorithms and specialized hardware accelerators has become essential to manage and analyze the rapidly expanding genomic datasets effectively.

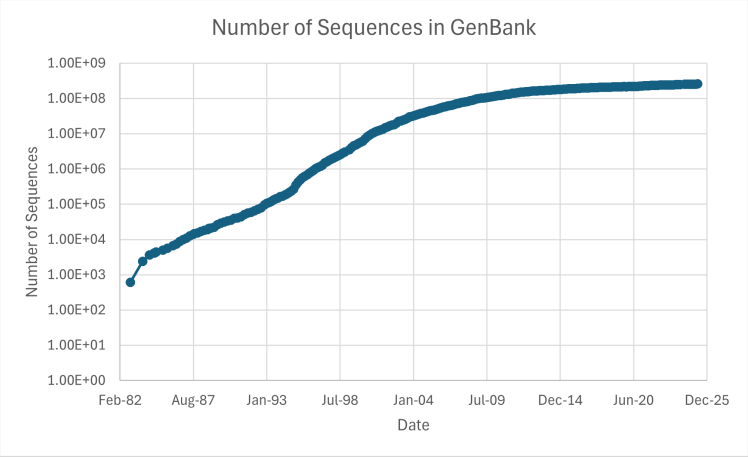


Figure 1: Number of sequences in the GenBank database. Adapted from the National Library of Medicine[1].

Smith-Waterman is an important algorithm in computational genomics, designed specifically for local sequence alignment. Its purpose is to identify the most similar regions between two biological sequences, such as DNA, RNA, or proteins, by comparing them and optimizing a similarity score[2]. Unlike global alignment methods, which attempt to align entire sequences end-to-end, Smith-Waterman focuses on finding the best matching subsequences, making it especially powerful for detecting conserved functional domains, mutations, or evolutionary relationships within larger, otherwise unrelated sequences.

Smith-Waterman is often used on its own to find localized similarity between two sequences, but it also serves as a key component in more complex algorithms, such as Partial Order Alignment (POA), where it helps to align sequences against graphs rather than just other sequences[3]. Smith-Waterman has notable challenges however: it is an $O(N^2)$ complexity algorithm, making it unwieldy for long sequences or large datasets. While Smith-Waterman is highly parallelizable in theory, the data dependencies between neighboring computations introduce a level of complexity that makes efficient parallel implementations non-trivial. Furthermore, it is not a computationally dense algorithm, meaning that performance bottlenecks often arise not from a shortage of compute resources, but from memory bandwidth limitations. As a result, achieving high performance, especially at scale, requires careful optimization of both algorithmic structure and hardware architecture.

To address these challenges, FPGAs (Field-Programmable Gate Arrays) have emerged as an attractive platform for accelerating Smith-Waterman. FPGAs offer the ability to custom-design parallel hardware pipelines that can better manage the fine-grained dependencies and memory access patterns inherent to the algorithm, leading to significantly improved throughput and energy efficiency compared to general-purpose CPUs and even GPUs. However, development on FPGAs is traditionally done using hardware description languages (HDLs) such as Verilog, which, while providing fine-grained control over memory utilization, parallelism, and timing, also significantly increases development time. Designing in Verilog requires developers to manage low-level details like external memory interfaces and to create cycle-accurate designs, making large-scale changes cumbersome and time-consuming.

To streamline FPGA development, High-Level Synthesis (HLS) tools have emerged as a solution. HLS allows developers to describe hardware behavior using high-level languages such as C or C++, dramatically reducing development time while still enabling generation of efficient hardware. Although HLS can abstract away many of the lower-level design concerns, achieving optimal performance often still requires careful tuning and understanding of the underlying hardware.

In this report, I will introduce how to implement Smith-Waterman, go over some software baselines, and explain my design space exploration of FPGA implementations. Afterwards, I will go over my testing strategy, and then end with the evaluation of my design. With my largest and most optimized FPGA implementation, I am able to reach calculation times, at worst, 2x as long as the off-the-shelf baseline. However, my implementation is not as energy efficient as the fastest CPU design, while being roughly as efficient as the FPGA alternative. HLS has shown to have the ability to quickly create designs with comparable performance to baseline CPU / GPU implementations, but the lack of control can lead to frustrations when optimizing, due to an inability to control resource usage, and see limiting parameters such as routing and timing analysis.

2.1 Smith-Waterman

The Smith-Waterman algorithm is used to find the best matching sub-segments between two sequences — like DNA, proteins, or strings of characters — by comparing all possible local alignments and scoring them based on how well they match. Unlike global alignment, it does not try to align the sequences from beginning to end but instead identifies the region with the highest similarity.

There are two variations of Smith-Waterman, linear gap penalty and affine gap penalty. This design project focuses on linear penalty. To identify sequence alignment, Smith-Waterman requires three different parameters: match bonus, mismatch penalty, and gap penalty (gap affine requires an extra parameter: gap extension). Match bonus is the increase in the similarity score when the two sequences have a matching base at that index and is positive. Mismatch penalty is the opposite; it is the decrease in the similarity score when there is a difference. This represents a mutation of a base (changing from one to another). Gap penalty is the decrease in the similarity score when a base is added to or deleted from one sequence. The two penalties are negative values. This gap penalty is linear, the cost of deleting two contiguous bases is the same as deleting two separated bases. In affine gap penalty, there is a larger base penalty for creating a gap, but a lower penalty for extending a gap.

		A	A	C	G
	0	0	0	0	0
A	0				
A	0				
T	0				
C	0				

Figure 2: An empty Smith-Waterman matrix before traceback. Sequence 1 refers to AACG, sequence 2 refers to AATC.

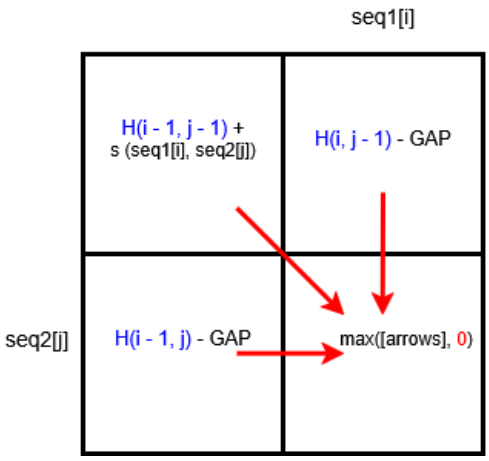


Figure 3: Depiction of the calculation of a single index. s() evaluates to either MATCH or MISMATCH, depending on if seq1[i] == seq2[j]. Sequence 1 is placed at the top of the matrix, and sequence 2 on the left.

Smith-Waterman consists of two primary steps: scoring and traceback. Scoring refers to creating a scoring matrix, which is a large 2d matrix. The matrix is of size A+1 by B + 1, where A and B are the lengths of sequence 1 and 2. Each column and row are associated with a single base, except for the first column and first row, which are not associated. The first column and row are set to 0. Each index is then calculated by taking the maximum of the following:

- The value of the index above, plus the gap penalty
- The value of the index to the left, plus the gap penalty
- The value of the index to the upper left, plus the element similarity score. This score is either the match bonus if the two sequence elements associated with the index are the same, or the mismatch penalty if they are different.
- The number 0.

Figure 3 above displays this in a visual way. The entire matrix is filled out in this manner. Note how each index needs the indices above, left, and above left to be calculated. Thus, each index actually needs every index in a rectangle where it is the bottom right corner to be already calculated in order to find its own value. This is an important consideration for parallelizing Smith-Waterman.

Traceback is a sequential way of finding the most aligned region. First, the index with the greatest value is selected. Then, go backwards to the index from which the score was calculated. For example, in the fully completed traceback matrix below (Figure 4), the traceback starts with index 7, and then goes to index 4, from which the value of 7 was derived. This repeats until the number zero is reached.

To actually generate alignment strings, two empty strings are created. For every diagonal step in traceback, a character is added to each alignment string based on the bases associated with the index that the traceback step started from. For example, using Figure 4, the step from the index with value 7 to the one with value 4 would result in both alignment sequences adding the character 'C', since that index is associated with a column C and a row C. An upwards movement represents an addition to sequence 2 of the associated character, while sequence 1 gets an empty spot (usually represented by a dash or hyphen). The opposite is true for a horizontal movement: sequence 2 gets an empty spot while sequence 1 gets a character. After these alignment strings have been generated, starting from the largest index, the order is reversed.

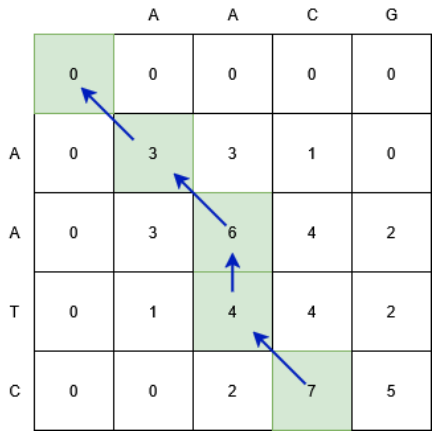


Figure 4: Traceback completed on a small Smith-Waterman matrix. The alignment score for this would be AATC and AA-C, when using parameters of Match = 3, Mismatch = -3, and Gap = -2.

In scenarios where there are ties, such as there being multiple of the maximum index, or multiple paths that lead to the same index on the traceback path, either value is a correct choice. Thus the result of a Smith-Waterman alignment may have multiple correct alignments. My implementation of the algorithm takes the upper left most appearance of the highest value, and prioritizes diagonal movements over vertical ones, taking horizontal ones last.

While filling out a matrix is parallelizable, traceback is difficult due to the sequential nature of the algorithm. This will be explored in the alternative design section.

2.2 High Level Synthesis

High-Level Synthesis (HLS) is a design methodology that enables the automatic translation of high-level programming languages, such as C++, into hardware description languages (HDLs) like Verilog. This approach allows hardware designers to describe complex algorithms and system behaviors at a higher level of abstraction, significantly improving productivity, design exploration, and time-to-market. By abstracting away the low-level details traditionally associated with RTL (Register Transfer Level) design, HLS facilitates rapid prototyping, design reuse, and easier integration with software-driven workflows.

One of the key advantages of HLS lies in its ability to expose parallelism and pipeline opportunities that are difficult to express or optimize manually at the RTL level. Designers can leverage compiler directives and pragmas to guide loop unrolling, function inlining, and dataflow pipelining—techniques that are essential for achieving high throughput and low latency in hardware. In Figure 5, loop unrolling and pipelineing are used to speed up parallel and sequencial operations in a multiply-accumulate function. Furthermore, HLS tools typically offer performance estimation and resource utilization reports early in the design process, enabling efficient architectural exploration and iterative refinement before committing to synthesis or place-and-route stages.

However, HLS removes control from the designer. It is difficult to specify that an operation should be split over multiple cycles to improve the critical path. Furthermore, exactly what hardware is generated is also difficult to control. For example, the designer may wish to instantiate a systolic array, but HLS may not schedule the processing elements to pass data systolically. For users from a software background, it takes some adaptation, understanding how an operation that may be trivial would require hardware resources in HLS. For example, accessing multiple elements in an array is trivial in software, but requires either a large section of block ram (BRAM) or many smaller registers, depending on the desired latency.

```
1     mac_array(op1[], op2[], seed) {
2         mulbuffer[];
3         output = seed;
4         for (i = 1; i <= SIZE; i++) {
5             #pragma HLS UNROLL
6             mulbuffer[i] = op1[i] * op2[i];
7         }
8         for (i = 1; i <= SIZE; i++) {
9             #pragma HLS PIPELINE
10            output += mulbuffer[i];
11        }
12        return output;
13    }
```

Figure 5: Pseudocode showing an example of a multiple-accumulate function in HLS. The multiplication is parallelized in hardware with UNROLL, while the accumulation is pipelined with PIPELINE. The multiplication output is also buffered. Some formatting such as datatypes are omitted for brevity.

2.3 Literature Review

In the course of this design project, I referred to several papers when considering optimization strategies. In order to get a fast baseline, I used the SIMD Banded Smith-Waterman implementation by Zhao et al[4]. Their work serves as an efficient baseline to compare my implementations against. While Banded Smith-Waterman does less work than my Smith-Waterman implementations, it is possible to attempt to compare them on even ground, which I will elaborate on in the baseline implementation section.

When this project was beginning, I was considering implementing the Partial Order Alignment (POA) algorithm, a sequence to graph alignment variation of Smith-Waterman. The SeGraM paper by Cali et al[5]. describes how they accelerated POA on an FPGA, using a significant amount of preprocessing steps to improve performance. I attempted to use some of these steps, but was not able to implement them in the final product.

The GenDP paper by Gu et al[6]. elaborates on their systolic array implementation of several sequencing algorithms, including both Smith-Waterman and POA. Their implementation, which had complex PEs with internal scratchpads and able to be rearranged in reconfigurable manners, was integral in showing me the strengths and weaknesses of a systolic array. Furthermore, their usage of a CPU / FPGA combo to calculate POA, with the CPU handling the long edge rare dependencies, showed to me the difficulty of accelerating POA.

The Allo paper written by Chen and Zhang[7] was important in understanding the first steps in FPGA optimization. While I did not use Allo in the end product, I utilized it in the beginning of my project, using it as a fast way to learn how different optimizations could affect my overall performance. I could easily evaluate the effectiveness of unrolling functional units, or adding buffers at critical steps, without the requirement of understanding the full HLS toolchain.

There is also a 2013 paper by Shah et al[8] that has a very similar design to mine. I found it when I was in the middle of optimizing the systolic array. Their design also has a 1D systolic array, as well as a left size buffer. However, they notably did not mention tiling at all, but had a pseudo tiling approach in one direction; that is, the PEs move to work on other columns after they are done. However, there is no tiling in the vertical direction, it is limited only by the size of the BRAMs (set to 500 in the paper). Furthermore, the paper also mentioned some extra steps that they had to use for BRAM control, which I did not have to do (or more likely HLS did for me).. I found it interesting, but restrictive due to the lack of tiling in the vertical direction, limiting sequence size.

Finally, the GMX paper by Doblas et al[9]. was an important inspiration for my final optimization steps. Their optimizations to data movement in the scoring and traceback steps resulted in a massive performance improvement when implemented on my design. I was not able to implement the other optimizations, such as banded Smith Waterman.

2.4 Acknowledgments, Collaborations, and Course Overlap

First and foremost, I would like to thank Professor Batten for his support and guidance throughout this project. His questions got me to reconsider my problems from different angles, and his enthusiasm was infectious. I could not have gotten so far as I have without his support.

I would also like to thank many of the PhD students (and visiting scholars) in the Computer Systems Laboratory. Niklas Schmelze was an invaluable help in understanding genomics algorithms, especially Partial Order Alignment (when this project was focused on POA). Max Doblas' work on Smith-Waterman ISA extensions was a crucial inspiration for my store optimization[9]. I used Hongzheng Chen's Allo framework[7] in the beginning of this project to understand how HLS works, and his explanations helped me understand how the HLS flow works. Andrew Butt was a great reference for other HLS related matters, especially in how HLS handles the FPGA memory hierarchy. Elton Shih helped me gather data on how genomics algorithms are used today, and explained the contents of his and Max's paper.

Finally, I would like to thank the other MEng students working on HLS under Professor Batten: Weixuan Sun, Zhangxiao Huang, and Jikai Wang. I was able to solve many problems by bouncing ideas off of them, and seeing if they had experienced similar issues. The baselines for this design were created in CS 5220 Applied High-Performance and Parallel Computing, with my teammates Jay Chawrey, Shivam Thakkar, and Shaunak Umarkhedi.

2.5 AI Acknowledgment

I used ChatGPT 4.0 Mini and Claude 3.7 Sonnet while working on this project. ChatGPT was primarily used for small random questions, while Claude was actually used for code. I found that neither was good at debugging HLS code, as they either fixated on how I indexed my arrays (which were controlled to avoid negative index accesses) or couldn't understand potential problems in the HLS Co-Simulation environment.

However, Claude was very helpful in modifying my host files: I created simple hosts that have hard coded datasets and correct communication with the device (or simulated device), and then had Claude handle adding file input and output comparison. Claude also helped implement the main function that executed the SIMD Banded Smith-Waterman implementation, but applying the same options as the host files.

Later on, ChatGPT was used for LaTeX questions, and cleaning up formatting on code blocks and tables. It performed remarkably well, allowing me to easily make my tables look better and simplified using LaTeX.

3 Baseline Designs

To accurately measure the effectiveness of the FPGA design, a baseline is needed for comparison. A baseline allows for a clear evaluation of improvements in performance, resource usage, and efficiency. To enable a thorough assessment, I created three different baseline designs. Two of these baselines incorporate parallelism, allowing for a fairer and more rigorous comparison against the FPGA implementation. By benchmarking against both sequential and parallelized versions, the evaluation captures the true advantages offered by the FPGA design.

3.1 Basic C Design

The basic design simply iterates over each index and calculates them sequentially. However, to optimize cache utilization, it uses tiling. Tiling breaks down a larger problem into smaller chunks. Every implementation, both base and alternative, uses tiling.

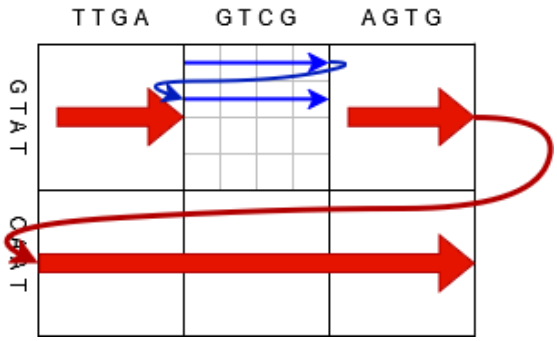


Figure 6: Explanation of tiling. Red arrows represent the order that tiles are calculated, while blue arrows represent the order that indexes are calculated inside each tile.

The baseline iterates over each tile, and within each index inside each tile, updating the maximum value tracker after every index calculation. Afterwards, the processor sequentially backtracks, starting at the maximum value. The resultant string is then reversed and then returned to the main function.

3.2 OpenMP Design

OpenMP is an Application Programming Interface (API) that allows for easy parallelization of tasks on CPUs. It supports C, C++, and Fortran. OpenMP is an implementation of multithreading, where one master thread spawns child threads which execute a task and then return to the main thread. The master and child threads all operate within a shared memory space, and can have selectively shared and private data. It supports running on a single processor, but multiple cores, as well as multiple processors, though the latter is architecture and setup dependent.

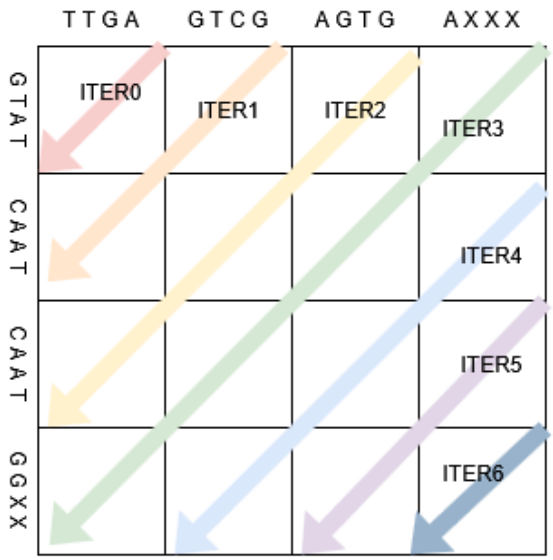


Figure 7: Example of wavefront parallelism on tiles. Each arrow represents one iteration of a loop, launching a thread for each tile that the arrow encompasses.

It offers easy parallelization of for loops with a pragma, as well as atomic operations to prevent race conditions, barriers to synchronize threads mid-execution, and data access control between threads. In this implementation, I will be simply using the for loop parallelizer, which spawns a thread for each iteration in a for loop. OpenMP also has schedulers which help efficiently allocate resources to threads that need them to optimize speed, and I will be using the base dynamic scheduler.

Technically speaking, Smith-Waterman is parallelizable by computing the element in each diagonal in parallel, then moving to the next diagonal. In practice, this would not use the cache as efficiently as possible, and thus be slightly inefficient with reusing data. However, the negative parts of this observation do not hold if the design was parallelized with tiles as the smallest unit, as each thread works on a tile, keeping the contents in its own cache, and is unaffected by the operations of other threads on other tiles.

This design has an average parallelism of N/2, where N is the number of tiles in the smallest direction.This method has the disadvantage of requiring many kernel launches, which each does a small amount of work, causing high overhead. This can be mitigated by increasing the size of the tile, so each kernel does more work, but this counters the efficiency gained by tuning the tile size to match the cache line size. This design shares the backtrace code with the basic C design, as adding parallelism to backtrace is incredibly difficult, if not impossible.

3.3 CUDA Design

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It enables developers to perform general-purpose computing on GPUs (Graphics Processing Units), extending the role of the GPU beyond graphics rendering to accelerate a variety of tasks. Originally, GPUs were designed primarily for rendering complex and detailed video game graphics, requiring the ability to perform a massive number of simple, parallel operations efficiently. As a result, GPUs are composed of thousands of relatively lightweight cores, each optimized for executing simple calculations in parallel.

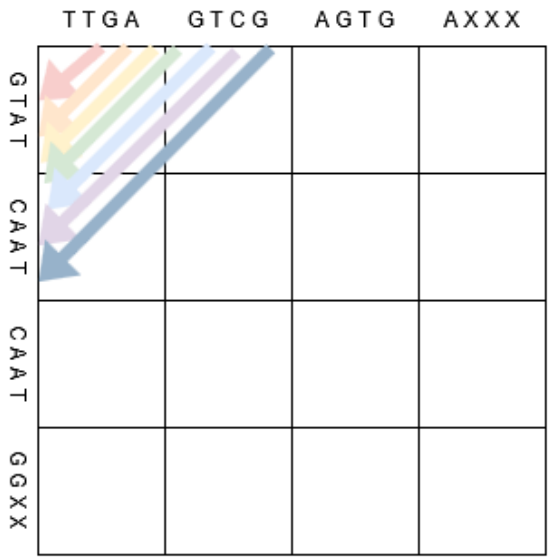


Figure 8: Wavefront parallelism in CUDA design. Each arrow represents one iteration of a loop, launching a thread for each tile that the arrow encompasses. Tiles are there for comparison purposes with the tiled design, there are no tiles in the CUDA design.

CUDA applications are written in C or C++ (or just recently Python), and run applications on large amounts of smaller CUDA cores. These cores are bundled into user-defined sizes of blocks, which each run the same kernel on different data, an application of Single Instruction, Multiple Threads (SIMT). As CUDA code is run on the GPU, CUDA kernels have memory that is separated from the host, and must use explicit calls to transfer data back and forth.

My CUDA implementation is a simple brute force design. Each kernel represents the computation of a single index. This kernel is run on a wavefront parallel style design, just like the tiled version, without the tiles. This does reduce cache performance slightly, but it reduces the amount of control flow required for execution, which is desirable since CUDA cores are optimized for computation, not control logic. The score matrix is stored in the VRAM, and is transferred to the host, to avoid doing the traceback on the GPU. This transfer takes a long amount of time, and could be optimized by doing traceback in CUDA. However, while this is a large drawback to my CUDA implementation, it is still very fast and capable of keeping up with other implementations.

3.4 Off-Shelf SIMD Banded Smith-Waterman

Banded Smith-Waterman is a variation of Smith-Waterman that assumes the given sequences are generally similar to each other. Once this assumption is made, the next assumption is that the highest value index is on or near the diagonal between the sequences (that is, from upper left to bottom right). Thus, the far corner indices are very unlikely to have the max index, and can be skipped. Banded Smith-Waterman creates a "band" that is a maximum distance that will be calculated from the center diagonal. The thinner the band, the less work will be done, but the higher the chance that true maximum index will be skipped.

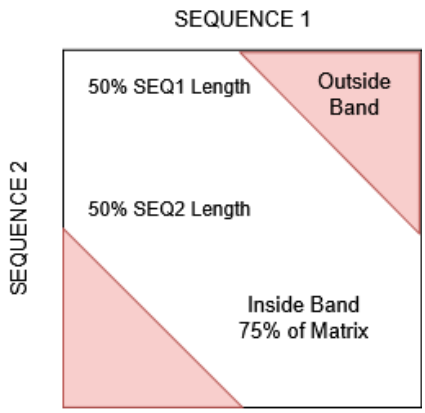


Figure 9: Banded SW example, with a band wise of 50% of sequence 2 length, assuming equal sequence 1 and 2 size. 25% of the scoring matrix is left uncalculated.

The implementation that I used is from Zhao et al[2]. Not only does it use a banded implementation, it also stores the data in diagonals, so that Single Instruction Multiple Data (SIMD) instructions can be used to increase parallelism. The band size is set to half the query string. Since my evaluation datasets have generally equal input sizes, we can say that their implementation computes 75% of the scoring matrix. Thus, calculation times for the SIMD baseline will be scaled by 33% higher to represent doing equivalent work with the other designs.

4 Alternative Design

For the alternative designs, I will start with the basic design, and qualitatively analyze its weaknesses and possible optimization strategies. I will then implement those optimizations, and look for more optimization opportunities

While the designs may differ in how they process the scoring matrix, they all share a similar overall flow. Each design segments the overall score array into tiles, and processes each tile sequentially. That way, an input of non-fixed (unknown at compile) size can be broken up into known size pieces, which can then be optimized. Processing a tile consists of: loading the boundaries from DRAM, processing the matrix, and then storing the completed matrix to DRAM. The circuit then backtraces starting from the maximum index.

To assist the FPGA, input sequences are always extended to be a multiple of the tile size, plus 1 (to account for a null terminator). While they are extended, there is no need to flush the padding with zeros; it can be left uninitialized. There is also an additional input specifying the sizes of the inputs, and the number of tiles needed for calculation. As a last step, since reversing a string of unknown (at compile) size is time-consuming on an FPGA, this step is left for the host.

Most of my optimizations will be targeting the tiled 'scoring' section in green (in Figure 10), as the backtrace is very sequential and difficult to optimize (but it is possible, as seen later).

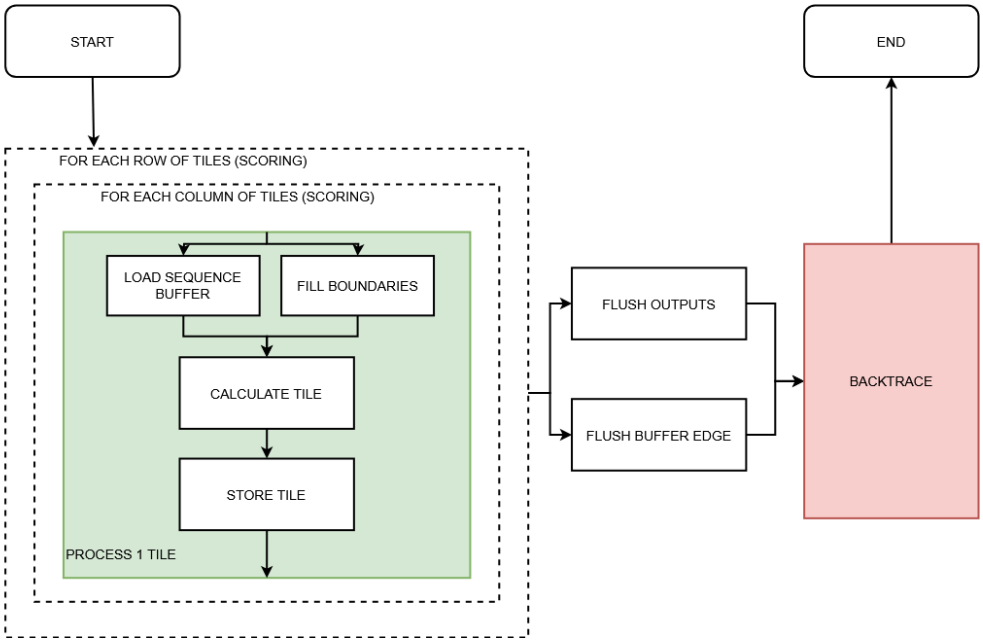


Figure 10: Flowchart of the general flow of Smith-Waterman FPGA Implementation. The tile calculation is denoted in green, and the traceback in red.

4.1 Loop Based Design

The first design is based on a simple nested for loop. Just like the tiles are chosen to be calculated, the loop based design iterates through the tile, calculating each index one by one. Thus, there is no parallelism, and the design is slow. Optimizations would target having any sort of parallelism or reuse of data other than the most recently calculated neighbor.

```
1     tile_function(score, buffer, seq1*, seq2*, seqsize[2], seq1buff*, seq2buff*) {
2         load_sequence_buffer(seq1, seq2, seqsize[2], seq1buff, seq2buff);
3         load_boundary(score, buffer);
4             for (i = 1; i <= TILE_DIMENSION; i++) {
5                 for (j = 1; j <= TILE_DIMENSION; j++) {
6                     calculate_index(i, j, seq1buff, seq2buff);
7                     check_maximum(score[i][j]);
8                 }
9             }
10        store_tile(score, buffer);
11        }
12    }
```

Figure 11: Pseudocode showing how the loop based function calculates the indexes in each tile. Datatypes have been removed for brevity. Technically, running this code would cause HLS to insert a pipeline in each for loop, so I manually disable pipelines for this design.

HLS offers a simple way to accelerate computation by using the PIPELINE pragma. By simply adding this to a loop, performance is increased as HLS attempts to increase parallelism in the design. However, how this is implemented is left out of the control of the user. There is no control, and no way to easily visualize the generated design, and thus while the resultant design will probably be high-performance, it is difficult to further optimize.

```
tile_function(score, buffer, seq1*, seq2*, seqsize[2], seq1buff*, seq2buff*) {
    load_sequence_buffer(seq1, seq2, seqsize[2], seq1buff, seq2buff);
    load_boundary(score, buffer);
    for (i = 1; i <= TILE_DIMENSION; i++) {
        #pragma HLS PIPELINE
        for (j = 1; j <= TILE_DIMENSION; j++) {
            #pragma HLS PIPELINE
            calculate_index(i, j, seq1buff, seq2buff);
            check_maximum(score[i][j]);
        }
    }
    store_tile(score, buffer);
}
```

Figure 12: Pseudocode showing the loop based design by applying pipeline to each loop.

4.2 Systolic Array Design

A systolic array is made up of several processing elements (PEs), connected together. They are connected in a way that allows for data to flow 'rhythmically' though the PEs. In a true systolic array, this means that only a few elements need to be connected to external datastreams, while most elements only input and output to other elements. A 1-dimensional systolic array can be used to calculate the score matrix. Each PE is assigned a row to calculate, and passes its calculated value to the PE calculating the row below it. This reduces the need for PEs to access shared memory, allowing more of the memory's time to be allocated to other tasks.

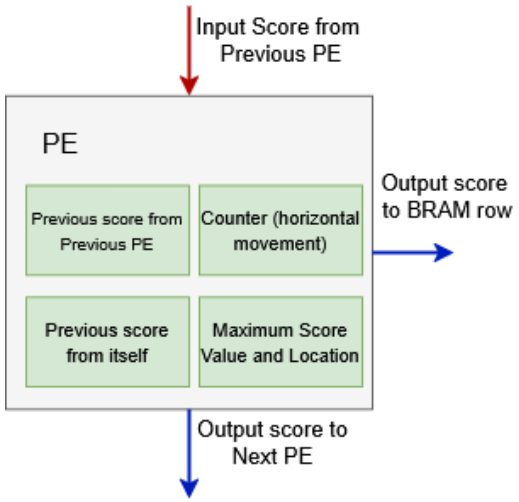


Figure 13: Diagram of a Single PE, showing its inputs and outputs. Internal registers are shown as green boxes. Values that do not change in a single tile (ex: tile location) are not shown.

As described in Section 2.1, every index in Smith-Waterman needs three values, the index to its left, upper left, and above. To avoid cross row memory access, each PE only accesses its own row, and is fed information about the row above from the previous PE. In Figure 14 below, the blue arrows represent the data flow from each PE, which stores data about the 'above' index. Each PE remembers its own previous value and the previous value sent from the above PE, a design similar to Shah et Al[6]. With this wavefront design, I have an average parallelism of N/2, as there are N PE's that work 50% of the time (PE_0 completes its work right as PE_N begins).

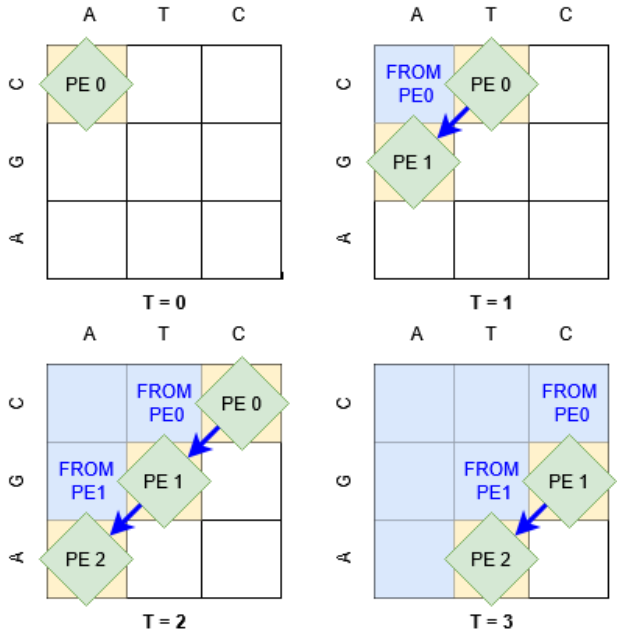


Figure 14: Small example of a 1D systolic array with 3 PEs calculating a 3x3 tile. Images are labeled with the iteration number below. Each PE moves horizontally across a row, calculating each value using the value calculated the previous cycle by the PE above it.

This design explicitly defines structures that HLS should generate, in contrast to the loop based design, where it is up to HLS to optimize the hardware. While theoretically this achieves maximum possible parallelism, and thus maximum calculation speed within a tile, the tile still needs to load and store values, which takes a significant fraction of the processing time.

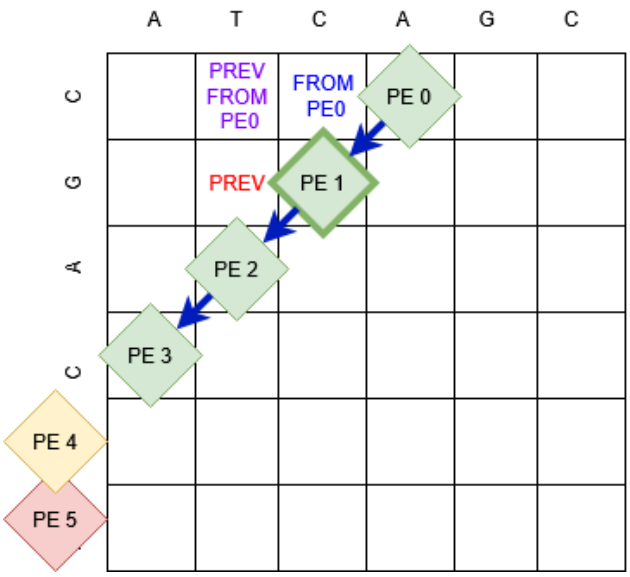


Figure 15: Diagram showing a 6x6 tile being calculated. PE 1, and the three indexes it uses to calculate its current value are highlighted, as well as the origin of each data point. PE 4 will start calculating on the next iteration, while PE 5 is still waiting for PE 4 to start.

4.3 Systolic Optimizations

As described in the last section, loading the boundary conditions and storing the resultant array is the next bottleneck to tackle. The amount of time spent loading boundaries can be reduced by adding a buffer in between tile calculations.

In Figure 16, green arrows represent fake dependencies; since the boundaries are zero, a read request to DRAM is not needed. A red arrow is a vertical dependency. Because the tiles are iterated row by row, it is not possible to buffer this dependency, as it is unknown when the tile below will be executed. The blue arrows can be buffered, because the tile to the right will be computed directly after the tile to the left. This should drastically reduce the amount of time to load boundaries, from >95% on the top edge tiles, to 50% for the inner tiles.

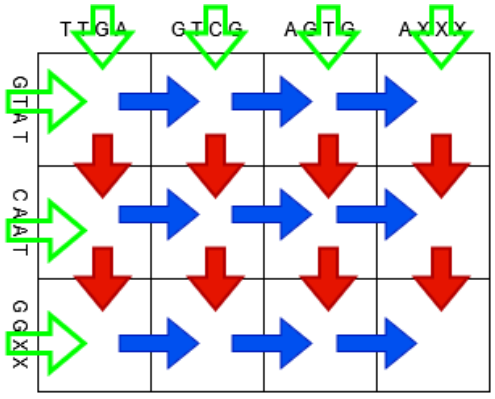


Figure 16: Breakdown of dependencies between tiles. Note each box represents a tile not an index.

While this optimization reduces the amount of time taken to load the boundaries, the load time still scales linearly with tile size. In order to fully see the speed gains from increasing the tile size, and thus increasing parallelism, the store step is the last bottleneck when it comes to computing the scoring matrix. However, unlike the boundary loading step, there is not a simple solution that would fix the issue. While every other component of processing a tile can be sped up using intelligent buffering or adding extra hardware, and thus becomes more efficient the larger the tile, storing every single calculated score index is an operation that scales quadratically with the length of a tile. Thus the only solution to this problem is to simply store less data.

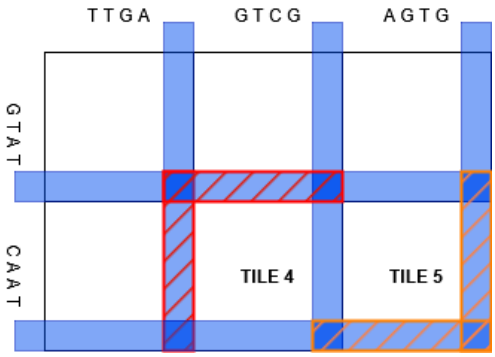


Figure 17: Breakdown of dependencies between tiles. Note each box represents a tile not an index.

By using a technique inspired from Doblas et al[7], each tile only stores what is required to calculate another tile: the last column and row. This is only 2*N stores, instead of N^2 stores per tile (where N is the tile dimension). Thus this reduces the overhead gained by increasing tile size. This also reduces the amount of DRAM buffer space required to store the results of each tile.

However, remember from the beginning of section 3 that calculating the tiles is not the only part of Smith-Waterman. After the maximum index is found through calculating all of the tiles, the path to reach that index must be traced back (the traceback step). In the previous designs, the entire score matrix was available in the buffer, and the traceback circuitry simply had to make memory requests to the buffer. Now, the tiles have to be regenerated from the stored boundaries.

Thus this design will also have an impact on the backtrace step, which has not been affected by any optimization up to this point. Qualitatively, the new backtrace method should be faster than the old one.

	Full Score Buffer	**Boundary Score Buffer**
Number of Memory Requests per Group	3	2N
Number of Traceback Steps per Group	1	$-1 \leq X \leq 2N$ (avg N)
Number of Memory Requests per Step	3	Avg 2

Table 1: Table showing the qualitative analysis of performing traceback using the full buffer or boundary buffer methods.

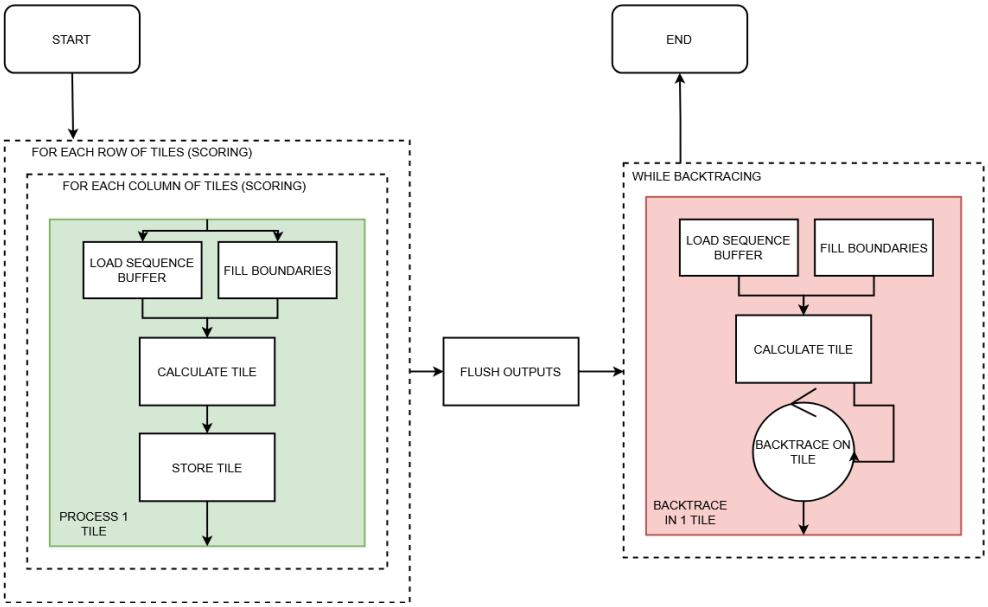


Figure 18: Diagram of the modified flow when using the optimized store design. The circular section represents how backtrace keeps going until it either reaches zero or exits the bounds of the tile. Note how the backtrace tile has very similar steps to the scoring tile.

4.4 Unused Optimizations

When considering the optimizations listed above, several others were considered but ultimately not implemented.

Chief among these ideas is double buffering, which involves having two score tiles. The PE array alternates working on each tile. When the tile is completed, it stores into the DRAM buffer. When the tile is about to start, it does a load from the DRAM. This does not decrease the actual latency between boundary load and result storing for one tile, but allows for loads and stores to be scheduled in parallel, reducing the number of cycles between each tile start. However, this design requires that the memory is able to handle simultaneous read-write requests. DDR4 (the DRAM on the FPGA PCB) is specifically half-duplex, which means that it cannot handle simultaneous reads and writes. There are, however, 4 banks of DDR4 ram on the PCB, and technically a design can be created that stores some of the buffer on each bank to allow for simultaneous reads and writes on separate regions of the buffer. However, I did not get to implement this design due to time constraints.

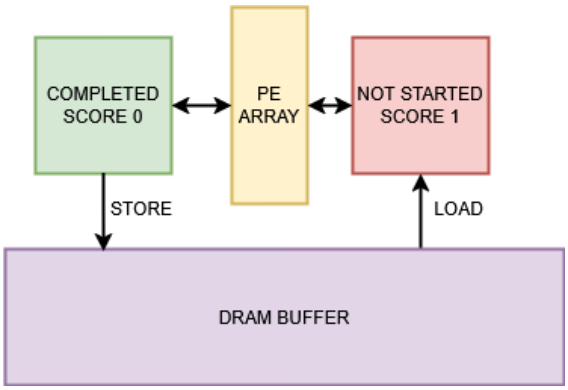


Figure 19: Example of double buffering. The green score 0 array is storing while the red score 1 array is loading, thus allowing for parallelization of loads and stores.

5 Testing Strategy

The Vitis testing flow consists of several steps, all of which come together to verify every aspect of the design, from the kernel to communicating with the host. Vitis handles the High Level Synthesis (HLS), converting C++ code into Verilog. Vitis can be run from either a command line interface (CLI) or a graphical user interface (GUI). For testing purposes, I will be using both; the CLI is used for mass testing a design, and the GUI is used for evaluating the specific effects of optimization strategies.

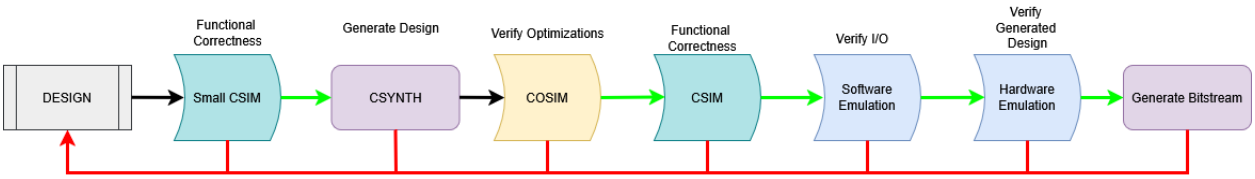


Figure 20: Overall testing flowchart. Designs start at the grey box, and move along the green path. A failure results in returning to the design to rectify the problem.

5.1 Csimulation (csim)

C-Simulation is a way of verifying the functionality of an HLS design. It simulates the behavior of the overall algorithm, allowing for functional correctness to be verified. This simulation executes the high-level source code as a standard software program, ensuring that the algorithm produces the expected outputs for given inputs. However, it does not simulate performance, because it abstracts away any hardware specific aspects like parallelism, and thus cannot simulate the effectiveness of pragmas or adding buffers.

I use multiple datasets to extensively test my algorithm, listed in Table 2. It is important to note that Smith-Waterman has multiple correct solutions, and thus my expected output may not be the same as the design output, but they are both correct. To mitigate this, all my baseline and alternative designs traceback in the same way, prioritizing diagonal, then vertical, and finally horizontal movements.

After altering any design, a subset of these tests is run to verify correctness before moving to cosim to verify the effectiveness of optimizations. If the optimizations are found to be effective, the full c-sim suite is run on the design to fully verify algorithmic correctness.

Test Case File	Number of Cases	Input Length Range	Description
Sequence Test	16	3 – 44	Randomized test cases made to test general backtrace.
Internal Alignment	16	5 – 50	Tests where one sequence contains the other.
Long Case	8	35 – 108	Longer tests with a variety of traceback directions.
No Alignment	9	0 – 143	Test cases where there is no alignment.
Perfect Alignment	11	5 – 8000	Inputs are identical. Also stresses long traceback handling.

Table 2: Test input types used to verify algorithmic correctness. These cases are executed automatically by a scripted workflow.

5.2 Co-simulation (cosim)

Co-Simulation (Cosim) is a way of verifying the correctness of the synthesized RTL design in an HLS flow. It simulates the generated hardware description (e.g., Verilog or VHDL) using a dedicated HDL simulator, while still being driven by the original C/C++ testbench. This allows for functional validation of the RTL output against the high-level source code, ensuring that synthesis preserves the intended behavior.

Unlike C-Simulation, cosim takes hardware-specific details into account. It includes the effects of scheduling, pipelining, and interface logic introduced during synthesis. This makes co-simulation more accurate in representing the final hardware behavior, though it is slower to run. It is particularly useful for verifying the correctness of pragmas and design optimizations, and it can also reveal issues that only occur after synthesis, such as protocol mismatches or incorrect timing assumptions.

However, cosim is incredibly picky, for lack of a better word. The same code, running on the same Vitis / Vivado version on the same server, may fail on the GUI but pass on the CLI. Furthermore, cosim wraps the entire testbench and the simulated RTL into the same executable. Any behavior that originates from the testbench that Vitis does not like causes a full end to cosim, even if the output of the simulation is correct. For example, adding file input to the cosim testbench causes the test to fail no matter what. For this reason, I use cosim strictly in the GUI to measure the effectiveness of optimizations, and use hard coded inputs to avoid the file I/O crash.

5.3 Software Simulation

Software emulation is very similar to c-simulation, they both shrink the kernel into a function level model, disregarding optimization and other pragmas. However, software emulation requires two files, the .xclbin and the executable. Note that the .xclbin can be three different things: a functional level model, an RTL model, or the actual bitstream which controls the FPGA configuration. Here it represents the FL model. Software emulation tests the host file and the way it communicates with the FPGA. It verifies that the FPGA would receive the correct input information during execution, and that the host is properly communicating with it. I use software emulation sparingly, primarily to verify that the kernel's external interface has not changed when optimizing it. C-simulation handles the functional correctness of the algorithm, so it is not necessary to use software emulation extensively.

To balance test coverage with efficiency, I typically select two test cases per input file when running

software emulation. These are usually the longest cases, as they are more likely to stress memory buffers and expose communication issues between host and device. This targeted testing ensures interface stability without incurring the cost of exhaustive emulation.

5.4 Hardware Simulation

Hardware emulation is a combination of the RTL simulation of cosimulation, and the host-device communication of software emulation. One of the key advantages of hardware emulation over cosimulation is its decoupling of the testbench and RTL. In traditional cosimulation, both the testbench and the synthesized RTL are wrapped into a single simulation environment. This tight integration can introduce idiosyncrasies that are not reflective of real-world host-device interactions. In contrast, hardware emulation allows the RTL to be driven by a standard software-based host application, mimicking the eventual deployment architecture more accurately and providing a clearer verification path.

Because the simulation operates at RTL granularity and mimics device-level communication, it is significantly slower than pure C-level simulation. This imposes practical limitations on the scale of testing that can be performed. Whereas C-sim can quickly validate a large set of datasets and edge cases due to its higher-level abstraction and execution speed, hardware emulation is best suited for a focused subset of tests. While these tests are generally a subset of existing test cases, care is taken that there is at least one test that is long enough to require the design to use multiple tiles.

Test Case File	**Number of Cases**	**Input Length Range**	**Description**
Sequence Test	8	3 – 44	Randomized test cases made to test general backtrace.
Internal Alignment	2	5 – 34	Tests where one sequence contains the other.
Long Case	2*	35 – 45	Longer tests if tile size is large enough to ensure multi-tile tests are exercised.
No Alignment	4	0 – 143	Test cases where there is no alignment.
Perfect Alignment	4	5 – 8000	Input sequences are identical. Also used to test very long tracebacks.

Table 3: Types of test inputs used for verifying RTL correctness. These are manually executed.

6 Evaluation

In evaluating my designs, I employed two complementary techniques: microbenchmarking and wall-clock testing. Microbenchmarking focuses on running small, targeted tests that isolate specific parts of the system. This allows for precise, quantitative analysis of the performance impact of individual optimizations or design changes. Through microbenchmarking, I could directly compare alternative designs under controlled conditions, making it easier to identify which optimizations offered meaningful improvements and which could be candidates for further refinement and scaling.

In contrast, by pushing my design through the flow, testing on the actual FPGA and using the real life execution time (wall-clock), I can compare the performance of my designs versus the baseline. However, this is time consuming, as generating the bit stream can take over 24 hours. Thus the wall-clock evaluation is only used on designs that have been shown to be optimized.

6.1 Microbenchmarking

Microbenchmarks are executed using the Vitis GUI. This allows me to use the timeline trace GUI feature, which offers an easy to visualize method of seeing how HLS schedules tasks, and the duration of each function. In the annotated figure below, the duration of each task is clearly noted and which tasks are running in parallel versus which are delayed by another task.

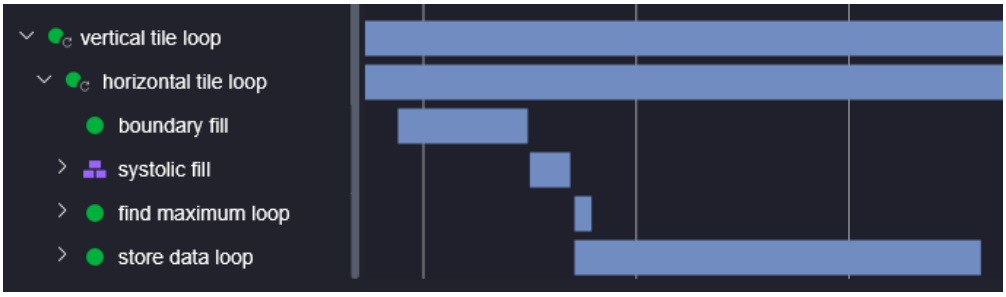


Figure 21: Example snippet of Vitis timeline showing the scheduling of a single tile. Each vertical mark represents 100 cycles. Each horizontal section can also be selected to show exact cycle start / end times.

For assessing the effectiveness of optimization strategies, the primary example in these examples is testing **ACTCCTATACC**, a 11 length sequence, versus **ACACCAGTACCCAAAACC**, a 18 length sequence, on a design that utilizes a tile size of 4 or 8. This is to keep simulation time low during rapid development. Area will not be evaluated in this section, as area is primarily a function of the tile size. The key metric that will be evaluated is the duration to calculate a tile. However, there are multiple types of tiles in the design.

In Figure 22, the green tile represents the corner tile. This tile does not depend on other tiles for boundary conditions, and thus has a very quick load cycle. The yellow edge tiles have one boundary that is not dependent on any other tile, but one boundary that does. They will take significantly longer to load, as they need to send read requests to DRAM. The red inner tiles require both boundary conditions to be read from DRAM, and thus will take twice as long as the yellow tiles. Thus this analysis will focus primarily on the inner tiles, as they make up the largest proportion of the calculation, and take the longest to process.

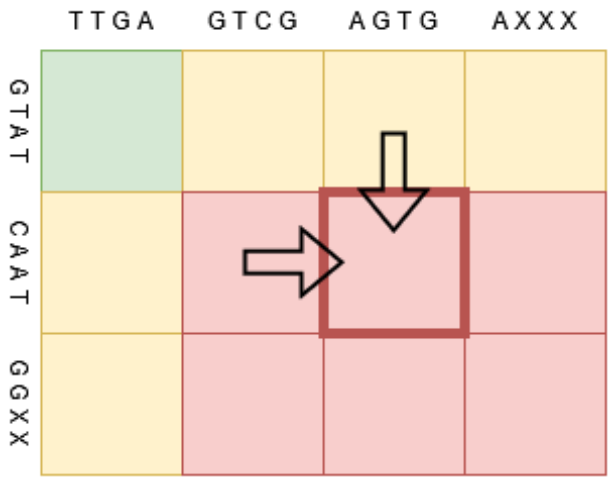


Figure 22: Example of the different tiles in a tiled implementation with 12 tiles. An inner tile and its boundary dependencies are highlighted.

6.1.1 Naive Loop

The first analysis is of the naive loop based implementation from section 3.1. As the tile size increases, the

Naive Loop	Tile Size = 4		Tile Size = 8	
	Cycles	% Total	Cycles	% Total
Boundary Load	126	56%	238	44%
Undefined Time	17	7%	21	4%
Matrix Calculation	28	12%	88	16%
Matrix Store	55	24%	191	35%
Total Tile Time	**227**	**100%**	**539**	**100%**

Table 4: Breakdown of number of cycles spent on each component of the tile calculation in the naive loop implementation. "Undefined Time" refers to time Vitis does not attribute to any function, likely due to setup overhead after reducing score matrix partitioning.

total calculation time (1x for 8x8 tile, 4x for 4x4 tile) decreases, showing how a larger tile is more efficient, as it spends less time loading boundary conditions. Note how the cycles spent to load boundaries scales by 2x, but the calculation and store steps scale by a factor >2x. This is because the boundary load is O(N) while the other two are O(N^2). Optimizing these steps is critical to decreasing latency.

6.1.2 Pipelined Loop

In comparison to the non-pipelined version, I see a 75-85% reduction in latency for the matrix calculation step. This shows the effectiveness of the pipeline pragma, especially considering that it required no more work than writing the pragma itself. Notably, the number of cycles is less than the amount of indices in the tile, showing that HLS is aggressively scheduling computations to achieve a high level of parallelism. However, I also do not know anything about how HLS actually implements this, without diving into the dense generated RTL.

The load and store operations are not affected, since the pipeline pragma is restricted to the calculation step. The 'undefined' time is also not affected, which lends more credibility to the assumption that it is time reserved for preparing the pipeline.

Pipelined Loop	**Tile Size = 4**		**Tile Size = 8**	
	Cycles	**% Total**	**Cycles**	**% Total**
Boundary Load	126	61%	238	51%
Undefined Time	18	9%	22	5%
Matrix Calculation	7	3%	15	3%
Matrix Store	55	27%	191	41%
Total Block Time	**206 (-21)**	**100%**	**466 (-73)**	**100%**

Table 5: Cycle breakdown for pipelined loop design. Numbers in parentheses represent change in latency compared to the previous implementation (the naive loop).

6.1.3 Base Systolic

Basic Systolic	**Tile Size = 4**		**Tile Size = 8**	
	Cycles	**% Total**	**Cycles**	**% Total**
Boundary Load	126	60%	238	51%
Undefined Time	17	8%	21	4%
Matrix Calculation	11	5%	19	4%
Matrix Store	55	26%	191	41%
Total Block Time	**209 (+3)**	**100%**	**469 (+3)**	**100%**

Table 6: Cycle breakdown for basic systolic design.

In Table 6, the results are very similar to Table 5 (and even slightly worse than), the results of the pipelined loop design. This means that the pipelined loop is creating at best, something similar to a systolic array, and at worst, a design efficient enough to compete with the systolic array. To test this, I tested both designs with a larger tile size.

Tile Size = 16	**Pipelined Loop**		**Basic Systolic**	
	Cycles	**% Total**	**Cycles**	**% Total**
Undefined Time	4	9%	4	20%
Matrix Calculation	41	91%	16	80%
Total Undefined + Calculation	**45**	**100%**	**20**	**100%**

Table 7: Comparison of cycle breakdown for Pipelined Loop and Basic Systolic architectures with tile size 16. The undefined time likely results from setup or synchronization effects.

The results show that the pipelined loop is not able to keep up with the systolic design at larger tile sizes. Thus the conclusion is that HLS is not instantiating a systolic array in the pipelined design, and that larger

sizes should use the systolic design. While the systolic array design is higher performance, the amount of work to create and verify this is orders of magnitude greater than the pipelined design.

6.1.4 Load Optimized Systolic

The load buffer is correctly reducing the amount of time it takes to load the tile boundaries. While the Vitis timeline trace makes it difficult to analyze the 4x4 tile design (due to optimizing out the loop), the 8x8 tile design shows a 67% reduction in latency on the load step. This is probably because the column loading used many non-consecutive load operations, and thus took longer than the single long load that is the top row. Since I have optimized this away, it had a large effect on the load function.

Load Opt Systolic	**Tile Size = 4**		**Tile Size = 8**	
	Cycles	**% Total**	**Cycles**	**% Total**
Boundary Load	*	*	75	75%
Matrix Calculation	11	8%	19	19%
Matrix Store	55	38%	191	191%
Total Block Time	**144 (-65)**	**100%**	**287 (-182)**	**100%**

Table 8: Cycle breakdown for load optimized systolic design. Note that the boundary load loop is optimized by HLS in the size 4 design, and is thus not visible as a discrete entity in the timeline trace.

6.1.5 Store Optimized Systolic

Load / Store Opt Systolic	**Tile Size = 4**		**Tile Size = 8**	
	Cycles	**% Total**	**Cycles**	**% Total**
Boundary Load	*	*	126	56%
Matrix Calculation	11	7%	19	8%
Undefined Loop	36	22%	21	9%
Matrix Store	**22**	13%	**94**	42%
Total Block Time	**164 (+20)**	**100%**	**226 (-61)**	**100%**

Table 9: Cycle breakdown for store optimized systolic design.

The store optimized design is more difficult to analyze, as the component functions of a loop calculation are broken up by dead space or unlabeled loops that HLS has generated without my prompting. This design attempts to combine the load optimization with the boundary storage optimization described by Doblas[7], but is less efficient than the load optimized design at smaller tile sizes. This is because Vitis decided to optimize out the left side buffer optimization from the previous design, so it no longer optimizes the load (this is why it is only store optimized). I have attempted to re-add this optimization, but Vitis refuses to do so. Figure 23 illustrates differences in speedup for the alternative implementations. Note that while it is slower at a smaller size, the store optimized design has a higher performance at larger tile sizes, and will continue to have a higher speedup, as it reduces a $O(N^2)$ step to $O(N)$ (where N is the tile size).

However, as mentioned in Section 4.3, the optimized store method also affects the traceback step. While I have qualitatively evaluated that it should be faster, I can now use the microbenchmarks to get quantitative

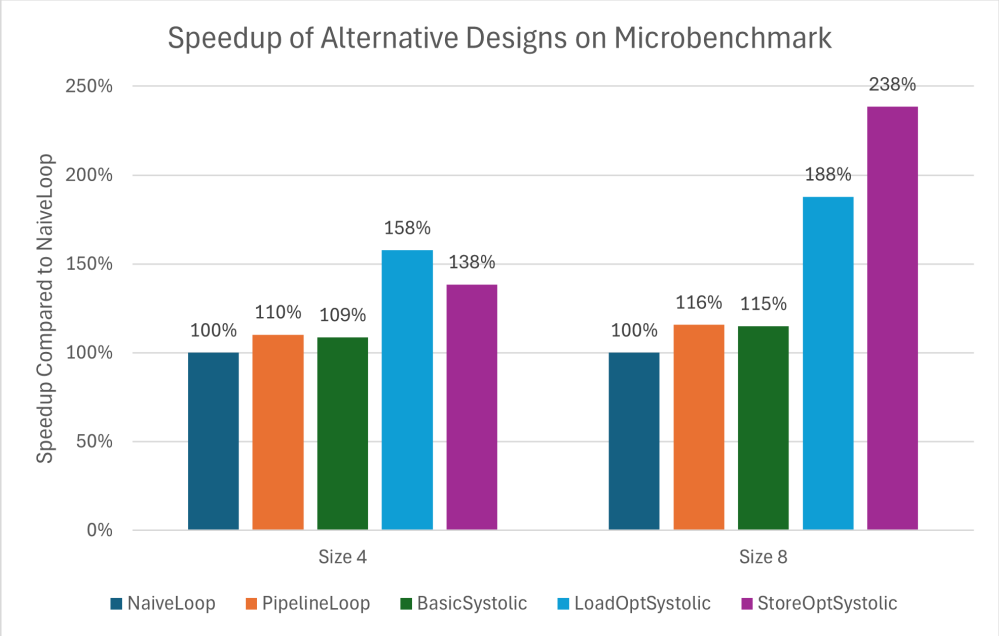


Figure 23: Summary of speedup of alternative designs compared to Naive Loop when run on the microbenchmark, using size 4 and 8 tiles.

results. As shown in Table 10, the two normal (element by element load) tracebacks take roughly the same amount of cycles, since this traceback method is not affected by tile size. However, the boundary method that the store optimized design uses is faster than the normal traceback at all tile sizes, and only gets faster as the tile gets bigger, as less tiles need to be loaded from DRAM. With this data in mind, the store optimized design is chosen for evaluation on the hardware using different tile sizes.

Traceback Type	Tile = 4	Tile Size = 8
Normal (used in Load Opt)	861 cycles	866 cycles
Boundary (used in Store Opt)	586 cycles	530 cycles

Table 10: Cycle comparison between Tile = 4 and Tile Size = 8 for different traceback types.

6.2 Evaluation Setup

Evaluation entails running the generated bitstream from my designs on hardware, and evaluating its performance versus a software baseline. All of these are run on the brg-zhang-xcel server.

The server is equipped with 48 Intel Xeon Silver 4214 Processors. This processor was released in 2019, on the architecture known as Cascade Lake, and is manufactured on the 14nm process. While I could not find the specific cache line size of this processor, the Cascade Lake architecture uses a 64 byte cache line in L1-L3 cache. Thus, the baseline design is optimized for this cache line size. OpenMP does not expressly limit itself to one processor, so technically the baseline design may utilize multiple processors. This will result in the execution speed being affected by the current server utilization. Baseline evaluation data is taken when the server is at a low utilization. The server also has 220 GB of RAM, which is enough to ensure that memory will never be a bottleneck.

Feature	Intel® Xeon Silver 4214 Specification
Total Cores	12
Total Threads	24
Base / Turbo Frequency	2.2 GHz / 3.2 GHz
L1 / L2 / L3 Cache Size	768 kB, 12 MB, 16.5 MB
Cache Line Size	64 Bytes

Table 11: Specifications of one of the 48 processors on the server.

The BRG team has access to a Nvidia 4070 GPU. While being marketed at consumers, the 4070 can also run CUDA applications, allowing for highly parallelized computations. Importantly, each streaming multiprocessor can only handle 1024 threads, so I am restricted to a maximum parallelism of 57k entries. The maximum VRAM, which holds data during CUDA operation will be a restriction if used to run massive datasets.

Feature	Nvidia 4070 Specification
CUDA Cores	5888
Streaming Multiprocessors	56
Base / Boost Frequency	1920 MHz / 2475 MHz
Memory Size	12 GB

Table 12: Specifications of the GPU available on the server.

All FPGA evaluation is done on a Alveo U250 FPGA. Notably, the FPGA has enough BRAM space to store a large Smith-Waterman matrix, but I will be emulating using a smaller device by constraining my tile size, and using DRAM as a buffer. This will result in my algorithm being restricted partially by the bandwidth between the FPGA chip and the PCB DRAM, but will allow me to run large datasets. Importantly, the DDR4 ram is half-duplex, and thus cannot support simultaneous read write operations.

Feature	Alveo U250 Specification
Lookup Tables	1,728K
Registers	3,456K
UltraRAMs	1,280
DDR Capacity	64 GB

Table 13: Specifications of the available FPGA.

6.2.1 Routing Congestion

When actually generating the bitstream, I ran into congestion issues. These indicate that the router, which finds paths to each logic element, is not able to create paths that connect blocks to each other. This routing failure appears in the high speed interconnect tiles. Through some trial and error, I found that my earlier attempt to increase the clock speed, which worked on smaller designs, was causing routing to fail on the larger designs. The difference in performance in small designs with faster clock speeds versus larger designs with lower clock speeds will be evaluated.

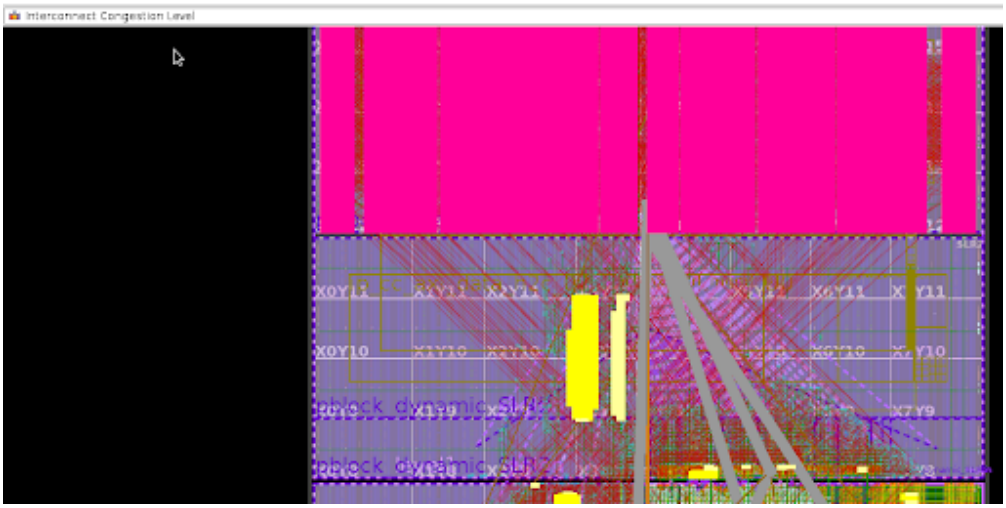


Figure 24: Image from Vivado congestion checker. The pink at the top represents a very high level of congestion (7) in the interconnects. This kills the place and route step.

6.3 Wallclock Time Evaluation

To test wall clock time, I created some evaluation datasets. Each dataset consists of two sequence inputs of roughly the same size, and very high similarity. This ensures that the traceback step scales roughly linearly with the input sequence size, so that there is a true correlation between input data size and calculation time. If this step is not taken, then a larger input size could have a short traceback step, skewing the calculation results.

Base Pair Length	Base	OpenMP	CUDA	SIMD*	Size 8*	Size 16	Size 32	Size 64
4	3.46E-06	2.13E-05	0.145	8.43E-06	2.31E-04	2.42E-04	2.77E-04	2.92E-04
8	7.70E-06	2.56E-05	0.144	8.57E-06	2.82E-04	3.02E-04	2.76E-04	2.99E-04
20	3.62E-05	7.76E-05	0.145	2.20E-05	3.22E-04	3.23E-04	3.11E-04	2.99E-04
100	7.24E-04	8.08E-04	0.145	9.82E-05	8.01E-04	7.57E-04	5.85E-04	4.55E-04
1k	0.0635	0.0111	0.158	0.0103	0.0540	0.0364	0.0185	9.27E-03
5k	1.60	0.187	0.215	0.240	1.270	0.878	0.420	0.207
10k	6.47	0.570	0.303	1.000	5.050	3.500	1.660	0.812
25k	41.8	3.93	3.08	3.55	33.1	21.8	10.4	5.02
75k	397	32.8	—	26.57	283	201	94.3	45.2

Table 14: Table of calculation time per implementation in seconds. Times less than 0.01 seconds are displayed in scientific notation. CUDA does not have a 75k score as there is not enough VRAM to execute the algorithm. SIMD baseline is scaled up by 33% from the actual runtime to account for being banded.

The CUDA time is constant at small sequences; this is because traceback is performed on the GPU in the CUDA implementation, which can account for the constant execution time at small sequence sizes. As the data and the graph shows, the larger the tile size, the lower the latency. This makes sense, because this maximizes the size of the systolic array, while also decreasing the amount of memory operations. Furthermore, the 8 size design, despite running at 400 MHz compared to the other FPGA designs running at 200 MHz, runs slower, showing that tile size dominates clock frequency. The time scaling of each implementation is also similar, around 9x-10x longer for 75k bp compared to 25k bp. This makes sense since Smith-Waterman is an $O(N^2)$ algorithm, increasing the sequence size by a factor of N, makes the amount of tiles computed scale by N^2. The tile size 64 implementation manages to stay on par or slightly faster than CUDA, OpenMP,

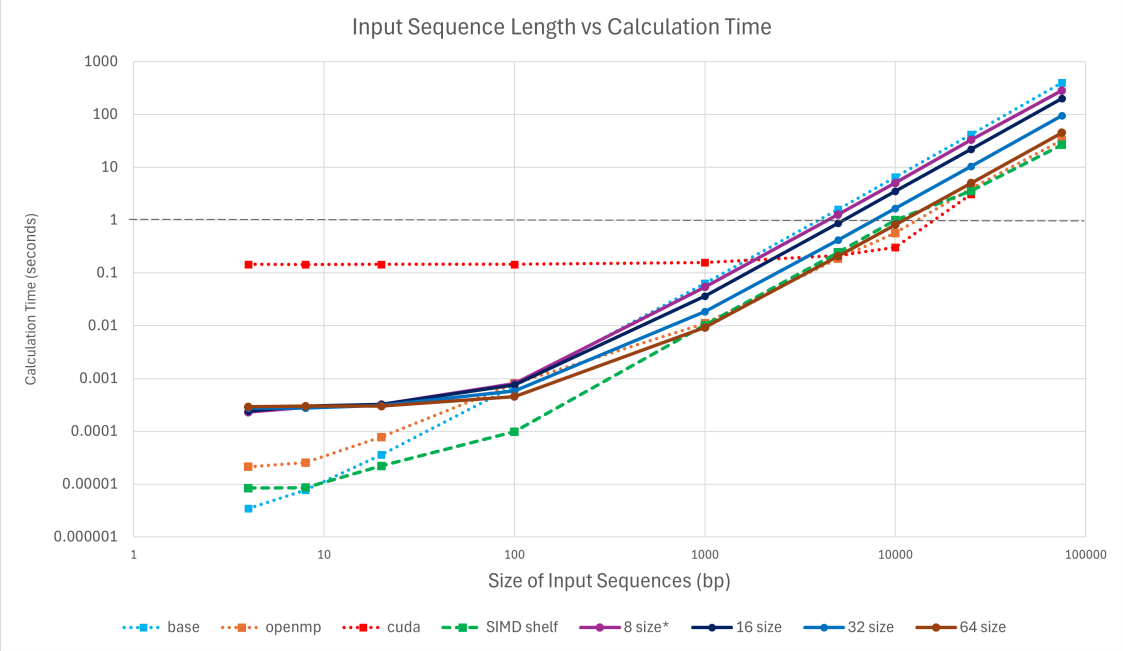


Figure 25: Data in Table 14 in a graph format. Both axes are logarithmically scaled

and SIMD, but starts to slow down comparatively as the size reaches 75k bp.

Adding debug information to the OpenMP implementation shows that there are threads running on CPUs labeled up to 49. A single Xeon Silver processor on brg-zhang-xcel only has 12 cores per processor, so OpenMP is distributing the work across multiple processors. As shown in the data above, the ability to process an entire diagonal of tiles at once using the massive parallelism of multiple processors allows the baseline to pull ahead of the FPGA at larger dataset. The SIMD implementation is single threaded, but can keep pace with OpenMP due to the lack of kernel launches, while having its own parallelism via SIMD instructions. For the FPGA designs to compete, larger tile sizes are needed.

6.4 Area / Resource Evaluation

Configuration	FF Usage	LUT Usage	BRAM Usage
Size 8 Store Opt	26039 / 1.5%	34416 / 4.0%	22 / 0.5%
Size 16 Store Opt	45292 / 2.6%	56120 / 6.5%	30 / 0.7%
Size 32 Store Opt	94696 / 5.5%	114873 / 13.3%	46 / 1.1%
Size 64 Store Opt	239612 / 13.9%	292364 / 33.8%	74 / 1.7%
Size 8 Load Opt	26889 / 1.6%	38638 / 4.5%	31 / 0.7%
Size 16 Load Opt	50413 / 2.9%	62381 / 7.2%	47 / 1.1%

Table 15: Raw and percent resource utilization on the U250 FPGA per design.

As FPGAs do not have an actual area amount to consider unlike ASICs, I will be using the resource utilization reports as a replacement. Table 15 shows the resource utilization of four difference sizes of the store optimized design, with two sizes of the load optimized design for comparison. Notably, the scaling in resource cost between each design is not linear. The size 16 store opt uses 60% more LUTs than the size 8 store opt , but the size 64 store opt uses 150% more than the size 32 store opt . This design is very light on the BRAMs and also uses no URAM, so memory/registers is not the restricting factor.

As stated in the section above, larger tile sizes are needed to compete with the parallel baselines. While running larger FPGA implementations is possible, it is difficult due to the complex routing. I attempted to synthesize a design with a tile size of 128, but the clock speed had to be reduced to actually allow timing to be met.

A simpler design like the base systolic (no memory optimizations) allows for larger tile sizes (such as 256), on a clock frequency of 200 MHz, but would lack the load / store optimizations, and thus may not actually have lower latency than a smaller tile design. The Table 15 contains size 8 and 16 load opt for this purpose. Surprisingly however, the load optimized designs have higher resource utilization than the store optimized design. While the resource utilization is higher, the congestion level is probably lower due to the less complex logic, possibly allowing for a higher frequency clock.

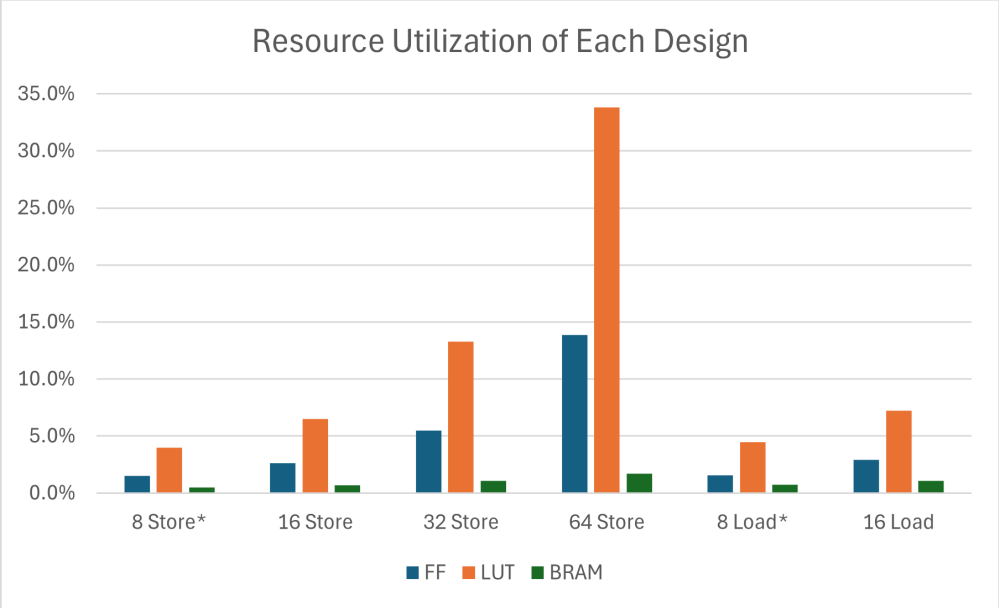


Figure 26: Percent resource utilization on the U250 FPGA per design.

6.5 Power / Energy Evaluation

It is difficult to perform an exact power evaluation, as I do not have precise power measurements of each design during runtime. Thus the approximate method would be to use the stated thermal design power (TDP) and multiply by the execution time. This would give us an estimate of the energy consumption. However, this also assumes maximum utilization of the silicon, which is not true. Just because the 64 size tile takes less time than the 8 size tile does not mean it consumes proportionally less power, it may in fact use more power, since more of the FPGA is utilized, but this is the assumption that has to be made. For the baseline, it is even more difficult to draw a strong correlation between power usage and runtime, as there are many power saving features on modern CPUs and GPUs.

The table above shows how the FPGA implementation is the least energy efficient out of all the designs, but this is only true with the assumptions above. From the resource evaluation in section 5.3, it is shown that the FPGA only uses 33% of its LUTs, and less than 15% of its other resources. Thus the FPGA energy estimation can be scaled down to somewhere around 20-25% of its TDP. However, this analysis also applies to the GPU and the CPU based designs; but I believe that they are using a larger percent of their resources compared to the FPGA. This is because the processor and code would be optimized for latency, and thus resources may be activated for short periods of time to compute a single small task, wasting power initializing a resource instead of waiting for a currently used one.

Implementation	Estimated Wattage	Estimated Energy
SIMD	200W	80W x 3.55 sec = 284 J
CUDA	200W	200W x 3.08 sec = 616 J
FPGA 64 Tile Size	200W	200W x 5.01 sec = 1002 J

Table 16: Approximate power calculation of each implementation, using TDP and calculation time on the 25k bp dataset. CPU power is ignored in CUDA as all the calculations are performed on the GPU, and thus CPU usage is marked as negligible.

In summary, it is difficult to gauge the energy usage of each design, but depending on the assumptions made, the largest alternative design can be competitive with the CUDA implementation, but is much more energy intensive than the SIMD implementation.

6.6 Conclusion

HLS is a powerful tool, allowing rapid prototyping of designs that have performance close to optimized CPU/GPU applications. It is especially useful in abstracting away considerations such as cycle accurate designs, something that would need to be considered if a HDL such as Verilog was used. It also allows for relatively quick changes to the core kernel, as C++ is much easier to modify than RTL.

However, in abstracting away many of the more tedious parts of RTL design, HLS also removes the ability to control some important factors, such as resource utilization and timing. In order to fully understand the reason a design fails to meet timing or routing parameters, a trip to the confusing HLS toolchain is needed. Furthermore, determining the critical path, a key component of RTL design, is very difficult in HLS, as cycle accurate control lies outside the hands of the designer in most circumstances.

In summary, HLS is best used when rapid prototyping is needed over the final efficiency of the design. This could be in a situation where there is little time to work on a design, or when the design requirements change so often that spending a large amount of time on a single design would result in a design that, while efficient, would be obsolete once completed. It could also be useful to verify optimizations as a proof of concept, before remaking the design in RTL.

References

[1] "GenBank and WGS Statistics," www.ncbi.nlm.nih.gov. [Online]. Available: https://www.ncbi.nlm.nih.gov/genbank/statistics/

[2] "Teaching - Smith-Waterman," *Uni-freiburg.de*, 2018. [Online]. Available: https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman

[3] "Understanding Partial Order Alignment for Multiple Sequence Alignment," *Github.io*, May 2015. [Online]. Available: https://simpsonlab.github.io/2015/05/01/understanding-poa/. [Accessed: May 15, 2025].

[4] 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*, vol. 8, no. 12, p. e82138, Dec. 2013, doi: https://doi.org/10.1371/journal.pone.0082138.

[5] D. Cali *et al.*, "SeGraM: A Universal Hardware Accelerator for Genomic Sequence-to-Graph and Sequence-to-Sequence Mapping," 2022, doi: https://doi.org/10.1145/3470496.3527436.

[6] Y. Gu *et al.*, "GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis," Zenodo (CERN European Organization for Nuclear Research), Jun. 2023, doi: https://doi.org/10.1145/3579371.3589060.

[7] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang, "Allo: A Programming Model for Composable Accelerator Design," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 593–620, Jun. 2024, doi: https://doi.org/10.1145/3656401.

[8] H. A. Shah, L. Hasan, and N. Ahmad, "An optimized and low-cost FPGA-based DNA sequence alignment — A step towards personal genomics," *PubMed*, Jul. 2013, doi: https://doi.org/10.1109/embc.2013.6610096.

[9] M. Doblas *et al.*, "GMX: Instruction Set Extensions for Fast, Scalable, and Efficient Genome Sequence Alignment," UPCommons (Polytechnic University of Catalonia), pp. 1466–1480, Oct. 2023, doi: https://doi.org/10.1145/3613424.3614306.