EC527 Final: Smith Waterman Algorithm

Seyed Reza Sajjadinasab, Bin Xu

Contents

1 Description of the Algorithm

The Smith-Waterman algorithm is a dynamic programming algorithm that is widely used in bioinformatics for sequence alignment. It was developed by Temple F. Smith and Michael S. Waterman in 1981.

The algorithm takes two sequences (typically DNA or protein sequences) and finds the optimal local alignment between them. Local alignment refers to the alignment of a substring from each sequence that maximizes the similarity score. This is in contrast to global alignment, which aligns the entire sequences.

The Smith-Waterman algorithm works by constructing a matrix (usually called the Smith-Waterman matrix or simply the dynamic programming matrix) in which each cell represents the score of aligning a particular pair of characters from the two sequences. The matrix is filled in by iterating over each cell, computing its score based on the scores of the neighboring cells, and selecting the highest score.

Once the matrix is filled in, the highest score in the matrix corresponds to the optimal local alignment score. The alignment itself can be obtained by backtracking from the highest-scoring cell to a cell with a score of zero, following the path of highest-scoring cells.

The Smith-Waterman algorithm is widely used in bioinformatics for tasks such as sequence database searching, gene and protein function prediction, and identification of homologous sequences. It is also used in other fields, such as speech recognition and computer vision.

1.1 Serial Code

In the serial code, the matrix is iterated row by row to update the score in three directions, which includes up, left and diagonal. It is relatively straightforward, but it can be computationally expensive for large sequences due to the nested loop structure used to fill in the matrix.

The main component parts contains:

Update Similarity matrix: The Similarity matrix function used to compute the score of each cell is based on the match/mismatch score of the characters being aligned, as well as the gap penalty. The score of each cell is the maximum of three possible scores: the score of the cell to the left plus the gap penalty, the score of the cell above plus the gap penalty, or the score of the cell diagonally above and to the left plus the match/mismatch score.

Traceback: Once the matrix is filled in, the optimal local alignment can be obtained by backtracking from the highest-scoring cell to a cell with a score of zero, following the path of highest-scoring cells. This is typically done using a recursive function that follows the path of highest scores.

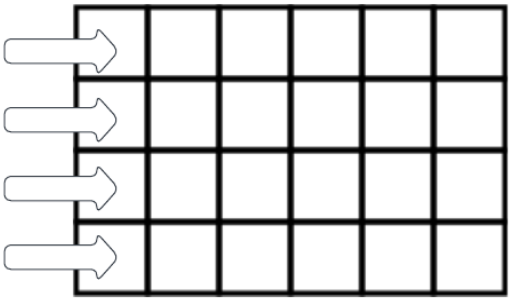

Figure 1. The graph shows the iteration order in the serial code is row by row

1.2 Serial Anti Diagonally

1.2.1 Preliminary considerations

The idea of serial anti diagonally verison is to iterate the matrix diagonally. The value of the current point depends on the up, left and diagonal. The way of iterating diagonally is more straightforward to understand the way of updating the score for every element of the matrix. The difficulty of this version is to find the start index and end index of every diagonal. The matrix is divided into three parts: left, middle and right. Then the index of up, left and diagonal are updated accordingly.

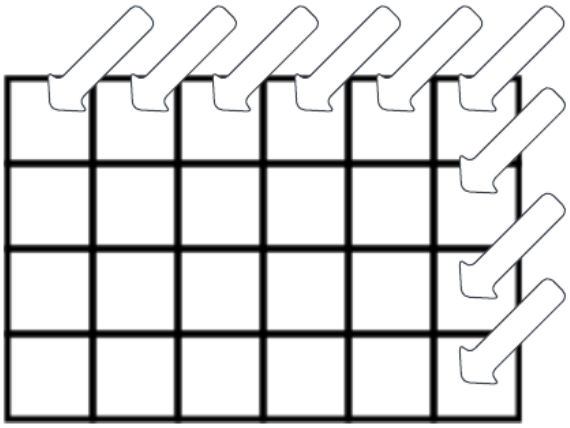

Figure 2. The graph shows the anti diagonal serial code is iterated by the diagonal

1.2.2 Modified code

In the below code, we find ind_u, ind_d, ind_l in three parts of the matrix, including left, middle and right.

```
long long int ind = 3;
long long int ind_u, ind_d, ind_l;
for (i = 2; i < m+n-1; i++) { //Lines
  long long int max_len;
  long long int ii,jj;
```

```
  long long int j_start, j_end;
  if (i<n){
     max_len = i+1;
  j_start = 1;
     j_end   = max_len-1;
     ind_u   = ind - max_len;
     ind_l    = ind - max_len + 1;
     ind_d   = ind - (max_len<<1) + 2;
  }
else if (i>=m){
 max_len = m+n-1-i;
     j_start  = 0;
     j_end   = max_len;
     ind_u   = ind - max_len - 1;
     ind_l    = ind - max_len;
     ind_d   = ind - (max_len<<1) - 2;
  }
else{
     max_len   = n;
     j_start      = 1;
     j_end       = max_len;
     ind_u       = ind - max_len - 1;
     ind_l        = ind - max_len;
     if(i>n)
        ind_d = ind - (max_len<<1) - 1;
     else
        ind_d = ind - (max_len<<1);
  }
```

In the below code, iterate every diagonal of the matrix to update the similarity score for the matrix.

```
for (j = j_start; j < j_end; j++) { //Columns
  if (i<m){
     ii = i-j;
     jj = j;
  }
  else{
     ii = m-1-j;
     jj = i-m+j+1;
  }
  similarityScore(ind+j, ind_u+j, ind_d+j, ind_l+j, ii, jj, H, P, max_len, &maxPos, &maxPos_max_len);
}
```

2 Scalar CPU version

In the CPU version, the AVX256 and AVX512 are used to optimize the serial code. 32-bit integers are used in the implementation of AVX.

2.1 AVX2 version

For having a vectorized version all we need was to translate the code presented in 1.2.2 and the actual similarity function to a vectorized version. Inside the similarity function, we defined 4 different __m256i registers. One for each element that needed to be read or written (Up, diagonal, left, and the actual element). However one issue here is that left and up registers almost overlap except for one element, but since we have enough registers in our CPU it wouldn't be a big deal.

2.1.1 Multithreaded version

We ran several threads to run each chunk of the antidiagonal simultaneously, but it was too slow and we didn't bring any result of that experiment here. A better multithreaded approach would be to find local alignments for different chunks of the longer protein that we are doing the alignment on.

2.2 AVX512 version

AVX512 is a deprecated version of intel AVX, and we could run it only on a skylake architecture. This version is the fastest version that we could implement. The downside is that we need to use an old architecture of Intel which is slower intrinsically (e.g. in terms of CPU frequency).

2.3 Code and Result

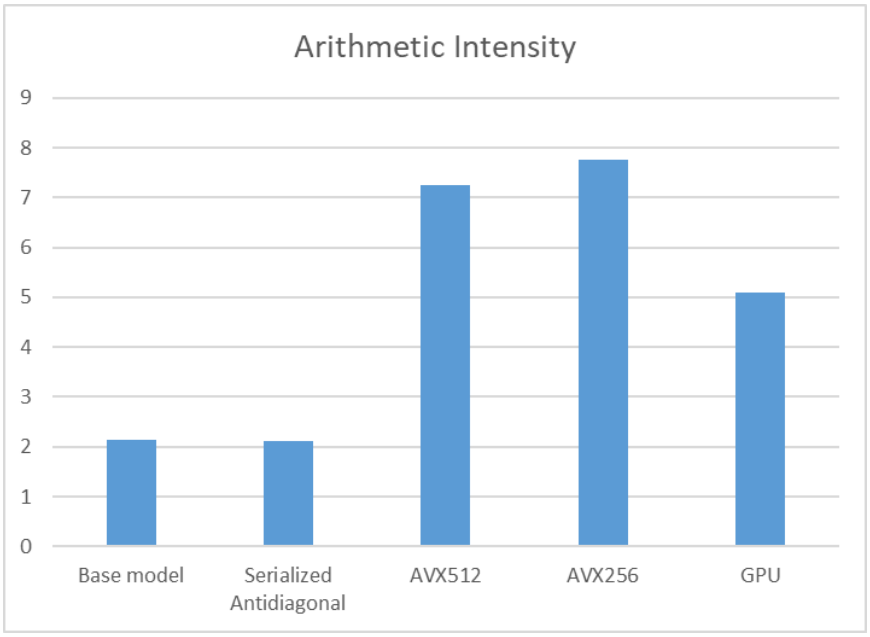

Figure 3. The graph shows the AI comparison among different version of SM algorithm

From the chart below, the AVX512 model gets 9X improvement. AVX2 model gets 5.5X improvement. This parallel processing capability makes AVX instructions ideal for scientific computing applications like the Smith-Waterman algorithm. By using the AVX instructions, the algorithm can perform many more operations per clock cycle, resulting in a significant speedup over the serial version.

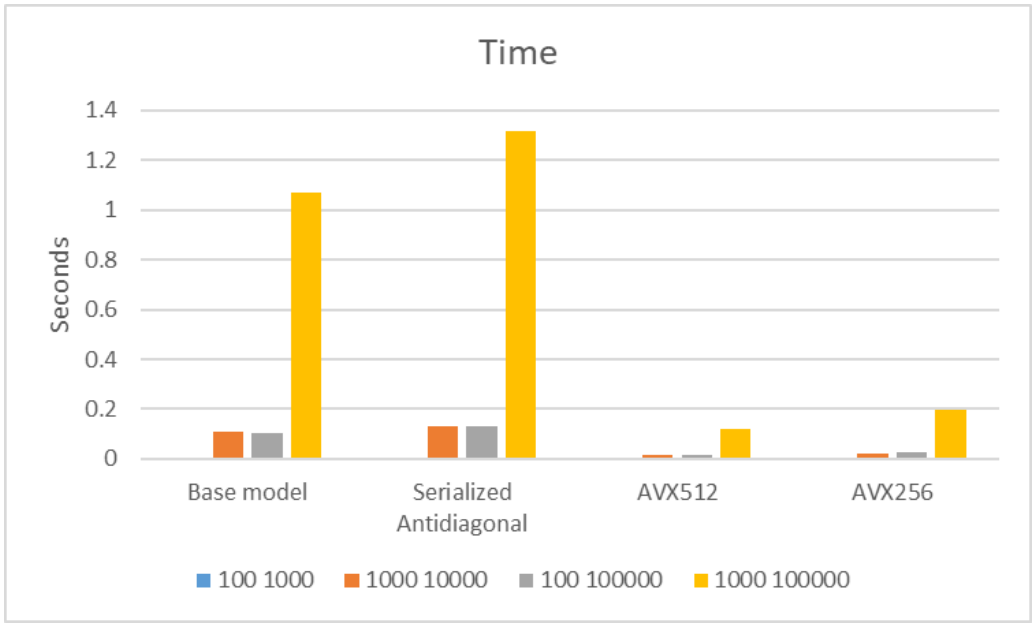

Figure 4. The graph shows the time comparison among different version of SM algorithm

Based on the benchmark we almost hit the CPU limits. However, the extension limit is approximate. Max memory bandwidth is 19.87 GiB/s. None of the operations are limited by memory.

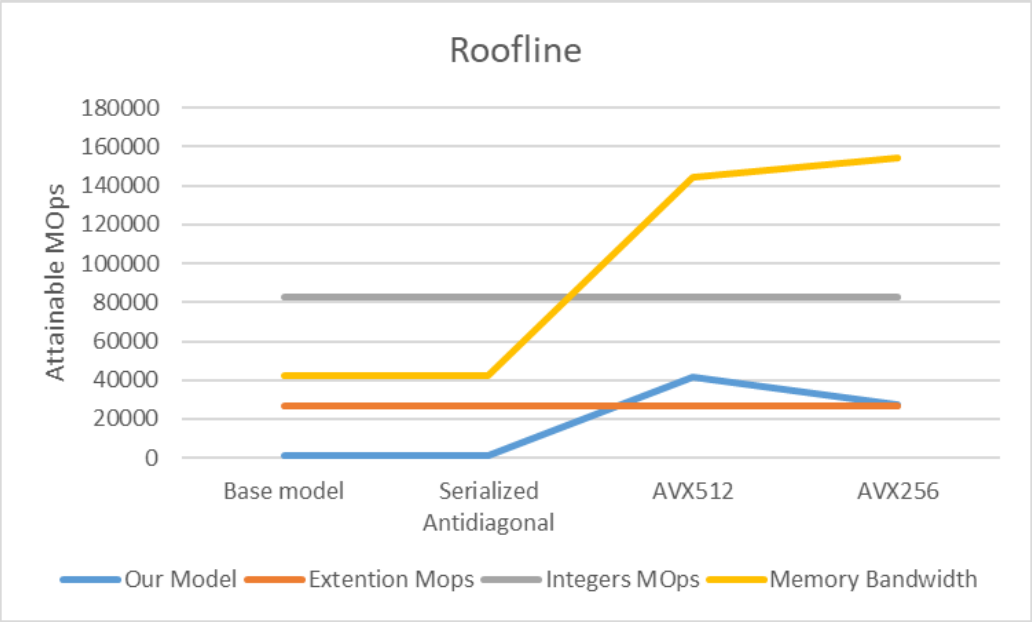

Figure 5. The graph shows the roofline between AVX version and serial version

2.4 Overall Comment

The comparison between the serial code, AVX512 version, and AVX2 version of the Smith-Waterman algorithm demonstrates the power of SIMD instructions in optimizing computational tasks. The difference in performance between AVX512 and AVX2 can be attributed to the increased capabilities of the former, allowing it to perform more operations in parallel. However, both versions demonstrate significant performance improvements over the serial version. Overall, the comparison emphasizes the importance of optimizing code for specific hardware architectures. By using SIMD instructions, it is possible to significantly improve the performance of computationally intensive tasks.

3 GPU version

3.1 Memory allocation

In the GPU version, the similarity score logic is set as a global kernel. In the kernel, the logic of finding the diagonal in which part of the matrix is inserted. The blockDim is set depending on the matrix size. The girdDim is set in terms of the length of diagonal and block size.
The following memory need to allocated:

1. Char *a: the memory space for storing horizontal character
2. Char *b: the memory space for storing vertical character

3. int *H: the memory space for matrix
4. int *P:the memory space for storing updated similarity score for matrix
5. maxPos: memory space for the position of max score in the matrix. The value is used for backtracking.
6. maxPos_max_len: memory space for the max score in the matrix. It is also used for backtracking.

3.2 similarityScore

The matrix is partitioned into three parts. In the global kernel, we calculate the length of the diagonal, starting point and ending point. These variables help me to find the current diagonal. using GPU to update the score on every element on the diagonal. The grid dimension is the length of diagonal / blockSize + 1. The block size is defaultly set as 100.
When calling the kernel, the cudaStreamSynchronize is used to address the synchronization issue in the GPU. In CUDA programming, kernels are launched on a device asynchronously. This means that the host thread does not wait for the kernel to complete before continuing execution. Instead, the kernel is queued for execution on the GPU, and control is returned to the host thread immediately. This allows the host to launch multiple kernels or perform other tasks while the GPU is executing the queued kernels. However, in some cases, it is necessary for the host thread to wait for a specific kernel or stream to complete before continuing execution. This is where cudaStreamSynchronize() comes in. By calling this function, the host thread can wait until all operations in a specific stream have completed before continuing execution.

3.2.1 cudaStreamSynchronize

In some cases, the results of one kernel may be required as input for another kernel. cudaStreamSynchronize() can be used to ensure that the first kernel has completed before launching the second kernel. Synchronization can incur some overhead, so it is important to use cudaStreamSynchronize() judiciously. By only synchronizing when necessary, the performance of the application can be improved.

3.3 Discussion of result

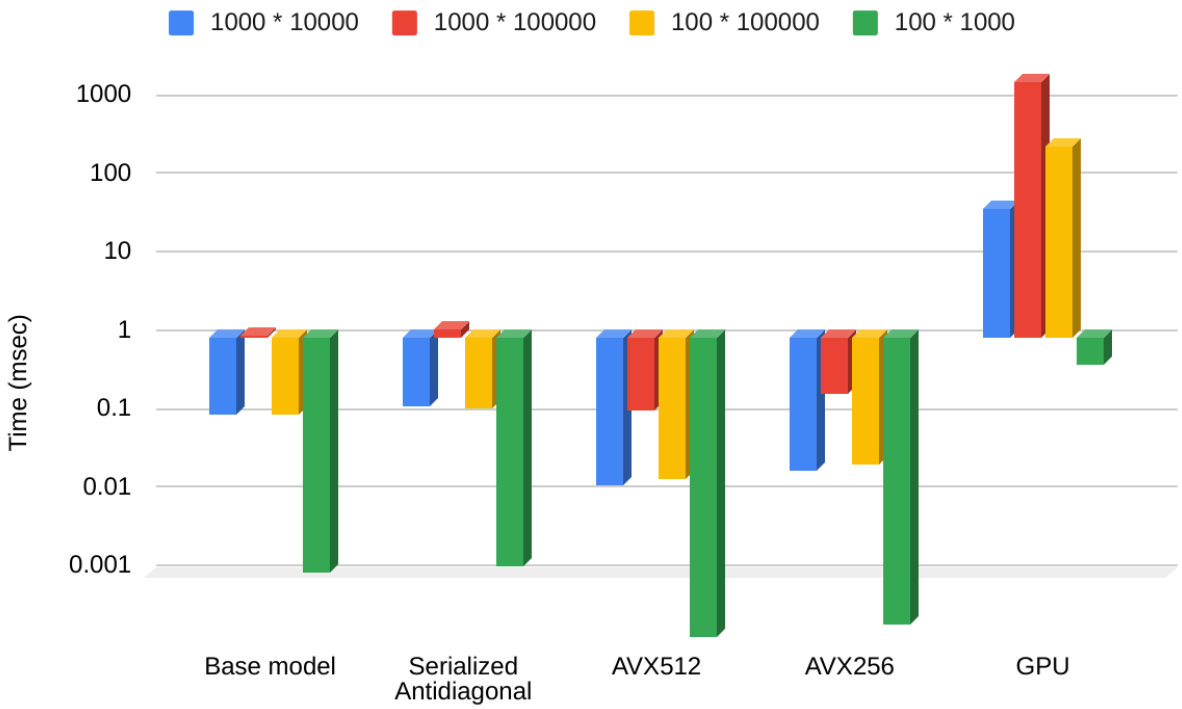

Figure 6. The graph shows the msec of different version of SM algorithm

The GPU version of smith waterman is much slower. At the beginning, we guess the problem is that copying the memory from CPU to GPU every iteration takes lots of time. After we integrate all the relative logic into the global kernel, the performance is obviously slower than the vector version. We think there is a synchronization problem in GPU implementation, before starting the next iterations or there should be a lot of data transferring between CPU and GPU in each iteration.

- **Memory bandwidth:** The Smith-Waterman algorithm involves accessing and updating large dynamic programming matrices. The performance of the GPU implementation can be limited by the memory bandwidth between the CPU and GPU, as well as the memory bandwidth between the GPU and its own memory. If the memory bandwidth is a bottleneck, the GPU implementation may be slower than the vector version.
- **Workload balance:** The workload in the Smith-Waterman algorithm is not always evenly distributed across threads or blocks. In some cases, some threads or blocks may finish their work much earlier than others, leading to idle time and reduced performance. Workload imbalance can be addressed by using load balancing techniques or adjusting the thread and block sizes.
- **Algorithm design:** The performance of the Smith-Waterman algorithm on a GPU can also be affected by the design of the algorithm itself. The anti diagonal version of serial code can be a problem to implement in GPU.

4 Closing Thought

The algorithm can be optimized for performance using hardware acceleration techniques such as AVX 256, AVX512, and GPUs. AVX 256 and AVX512 are vector instruction sets that allow for efficient parallelism on CPUs. By optimizing the algorithm to use these instruction sets, the performance of the Smith-Waterman algorithm can be significantly improved. However, the performance gains depend on the characteristics of the specific CPU and the workload.
GPUs can also be used to accelerate the Smith-Waterman algorithm by parallelizing the computations across many processing cores. GPU implementations can provide significant speedup compared to vector versions, but their performance can be affected by factors such as memory bandwidth, thread divergence, workload balance, and algorithm design.
Optimizing the Smith-Waterman algorithm for performance requires careful consideration of the hardware architecture, workload characteristics, and algorithm design.

In the future work:

- We implemented a multicore version both with openMP and Pthread. Neither was helpful. We can run it across different chunks of a protein instead.
- There can be some improvements over how we implements barrier in Cuda so that we can have a better performance
- We can increase AI even further for GPU by unrolling the function by a factor of 2 and we can have 80% improvement for the GPU version
- We can use short integer and have a saturation for maximum similarity points

5 Compilation instructions

5.1 Scalar Code

5.1.1 AVX256
To compile the GPU code, run:

```
gcc -O1 SWalgo_V2_AVX256.cu -o SWalgo_V2_256 -lgomp
```

5.1.2 AVX512
To compile the GPU code, run:

```
gcc -O1 SWalgo_V2_AVX512.c -o SWalgo_V2_AVX512 -lgomp
```

5.2 GPU Code

To compile the GPU code, run:

```
nvcc -O1 SWalgo_V2_GPU.cu -o SWalgo_V2_GPU -lgomp
```

- -lgomp: OpenMP library

Before compiling the code, the block_size can be set depending on the personal needs. After the compiling, adding the matrix row length and col length to run the program.

6 List of Code

The files included in the *.zip file include:

- SWalgo_V2_GPU.cu, which is the GPU version of algorithm
- SWalgo_V2_AVX256.c, which is the AVX2 version of algorithm. It can be runned on the most of cpu
- SWalgo_V2_AVX512.c, which is the AVX512 version of algorithm. AVX512 is a deprecated version of intel AVX, and we could run it only on a skylake architecture.
- SWalgo_V2_multithread.c, which is the multithread version of algorithm. SWalgo_V2.c, which is an anti diagonal version of smith waterman algorithm.
- SWalgo_V1.c, which is the serial version found in github.