De La Salle University, Manila
Computer Technology Department

Term 3, A.Y. 2023-2024

CEPARCO

INTEGRATING PROJECT
Implementation of the Smith-Waterman Algorithm in SIMT using CUDA

GROUP 3

Ambrosio, Carlos Felipe Q.

Arceta, Althea Zyrie M.

Mendoza, Antonio Gabriel

Tan, Jose Tristan T.

July 30, 2024

1. Project Description

Sequence alignment is a technique used to compare nucleic acid or protein sequences in order to identify regions of similarity between the sequences. The Smith-Waterman Algorithm is one such algorithm for sequence alignment, specifically local sequence alignment. Instead of comparing entire sequences, the algorithm looks for and compares segments of all possible lengths to identify similarities and obtain a measure of similarity (Smith et al., 1981). However, sequence alignment is computationally costly since its computing and memory requirements increase significantly as the database or queries increase, resulting in high execution time.

Researchers have tried a variety of methods to speed up the Smith-Waterman algorithm. One study employs reconfigurable computing architectures based on FPGA devices. This implementation has demonstrated performance gains of up to 28x above standard microprocessor-based methods (Storaasli et al., n.d.). Another study proposed a SIMD implementation, which achieved estimated execution speeds of up to 23.8x, rivaling FPGA implementations (Rudnicki et al., 2008).

With the advancement of hardware technology, GPUs provide new possibilities for improving algorithms. CUDA's parallel computing capabilities can improve the Smith-Waterman algorithm's computing efficiency. The proposed project aims to utilize the SIMT paradigm using CUDA to implement and parallelize the algorithm. Protein sequences will be the choice of input data to be compared and aligned instead of nucleic acid sequences (DNA or RNA). With that, the group will also utilize the BLOSUM substitution matrix instead of the simple substitution matrix to score the similarities of the protein sequences better. Lastly, since the linear gap penalty applies the same penalty to a single large gap as to multiple minor gaps, the group will utilize Affine gap penalties to designate insertion/deletion scores instead of a linear gap penalty. Unlike linear gap penalties, the affine gap penalties are gap length-dependent (Rotcha et al., 2018).

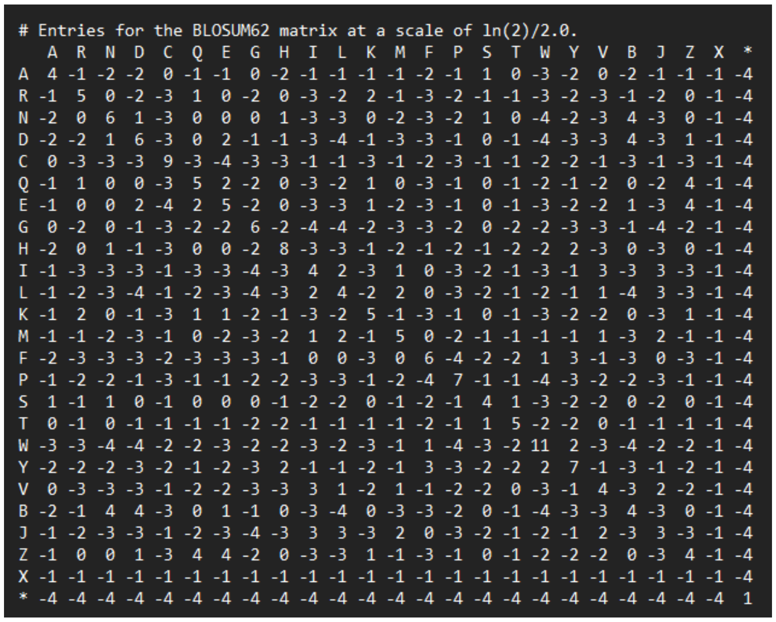
The group will implement and execute the algorithm in two separate kernels, one sequentially in C as a baseline and the other utilizing CUDA, accounting for the required changes to make efficient use of the parallelization present in CUDA. The respective latencies of both kernels are then recorded several times in order to verify the computing efficiency of both algorithms properly. The results will then be analyzed to validate whether or not the CUDA implementation will be more efficient than the C implementation, as well as to score the correctness of the algorithm itself.

2. Smith-Waterman Algorithm in CUDA

The Smith-Waterman Algorithm is comprised of 4 steps, namely: determining the substitution matrix and gap penalty scheme, scoring matrix initialization, scoring, and traceback. For this project, only the scoring part of the algorithm is to be parallelized and timed for comparisons.

a. Substitution Matrix and Gap Penalty Scheme

The substitution matrix and gap penalty scheme to be used are the **BLOSUM62** substitution matrix and **affine** gap penalty scheme with a gap open penalty value (v) of 5 and a gap extension penalty (u) of 1.



```
# Entries for the BLOSUM62 matrix at a scale of ln(2)/2.0.
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  J  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -1 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1 -2  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  4 -3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4 -3  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -1 -3 -1 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0 -2  4 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1 -3  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -4 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0 -3  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3  3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4  3 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0 -3  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3  2 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3  0 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -3 -1 -1 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0 -2  0 -1 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1 -1 -1 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -2 -2 -1 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -1 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3  2 -2 -1 -4
B -2 -1  4  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4 -3  0 -1 -4
J -1 -2 -3 -3 -1 -2 -3 -4 -3  3  3 -3  2  0 -3 -2 -1 -2 -1  2 -3  3 -3 -1 -4
Z -1  0  0  1 -3  4  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -2 -2 -2  0 -3  4 -1 -4
X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

BLOSUM62 Substitution Matrix

$$W_k = uk + v$$

Affine Gap Penalty Equation

b. Scoring Matrix

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ \max_{k \geq 1}\{H_{i-k,j} - W_k\}, \\ \max_{l \geq 1}\{H_{i,j-l} - W_l\}, \\ 0 \end{cases} \qquad (1 \leq i \leq n, 1 \leq j \leq m)$$

where

$H_{i-1,j-1} + s(a_i, b_j)$ is the score of aligning a_i and b_j,

$H_{i-k,j} - W_k$ is the score if a_i is at the end of a gap of length k,

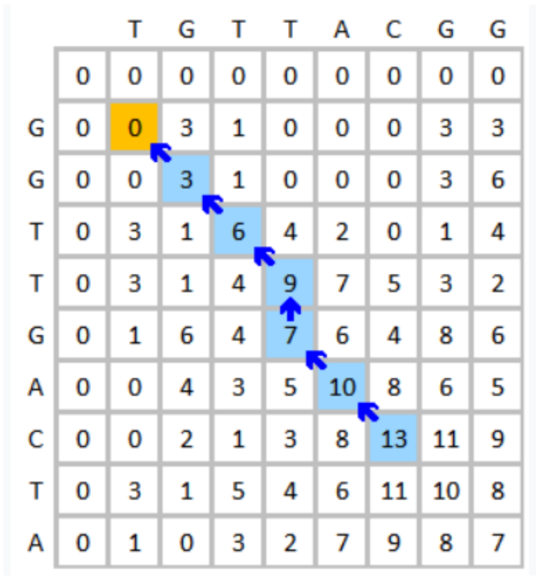
$H_{i,j-l} - W_l$ is the score if b_j is at the end of a gap of length l,

0 means there is no similarity up to a_i and b_j.

The ***mana* function** is responsible for the alignment scoring of the Smith-Waterman Algorithm. The algorithm follows a wavefront approach where the computation proceeds in diagonals across the matrix. This allows for parallel computation because each element on a diagonal can be computed independently of others on the same diagonal. This approach also ensures data dependencies are respected, as each cell depends on the values of neighboring cells (up, left, and the upper left diagonal).

c. Traceback

After the scoring matrix is fully populated, the ***traceback* function** is used to find the optimal alignment. This step is performed sequentially and is not parallelized since it involves a single recursive path through the matrix.

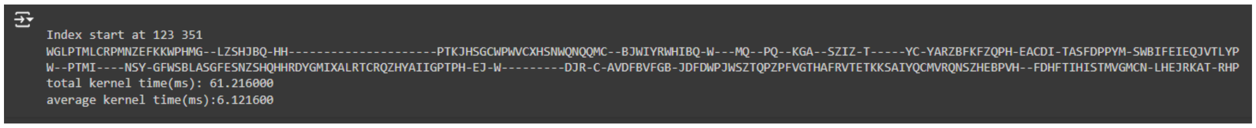


		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0	3	3
G	0	0	3	1	0	0	0	3	6
T	0	3	1	6	4	2	0	1	4
T	0	3	1	4	9	7	5	3	2
G	0	1	6	4	7	6	4	8	6
A	0	0	4	3	5	10	8	6	5
C	0	0	2	1	3	8	13	11	9
T	0	3	1	5	4	6	11	10	8
A	0	1	0	3	2	7	9	8	7

Sample Traceback

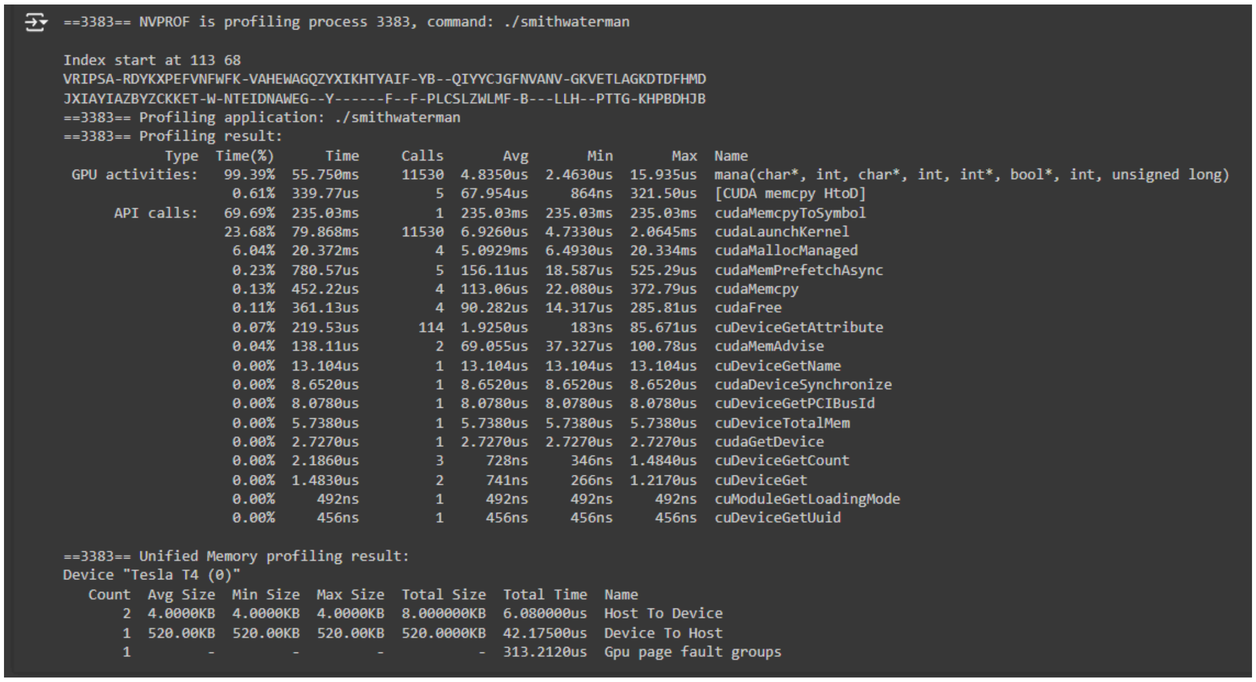
3. Execution Time Comparison

Both implementations are run 10 times and the average runtime is calculated. Below are some sample outputs, a table of the average runtime of the Sequential and CUDA implementation for different input array sizes, and a table containing speedup comparisons.



```
Index start at 123 351
WGLPTMLCRPMNZEFKKWPHMG--LZSHJBQ-HH---------------------PTKJHSGCWPWVCXHSNWQNQQMC--BJWIYRWHIBQ-W---MQ--PQ--KGA--SZIZ-T-----YC-YARZBFKFZQPH-EACDI-TASFDPPYM-SWBIFEIEQJVTLYP
W--PTMI----NSY-GFWSBLASGFESNZSHQHHRDYGMIXALRTCRQZHYAIIGPTPH-EJ-W---------DJR-C-AVDFBVFGB-JDFDWPJWSZTQPZPFVGTHAFRVTETKKSAIYQCMVRQNSZHEBPVH--FDHFTIHISTMVGMCN-LHEJRKAT-RHP
total kernel time(ms): 61.216000
average kernel time(ms):6.121600
```

Sample C Output

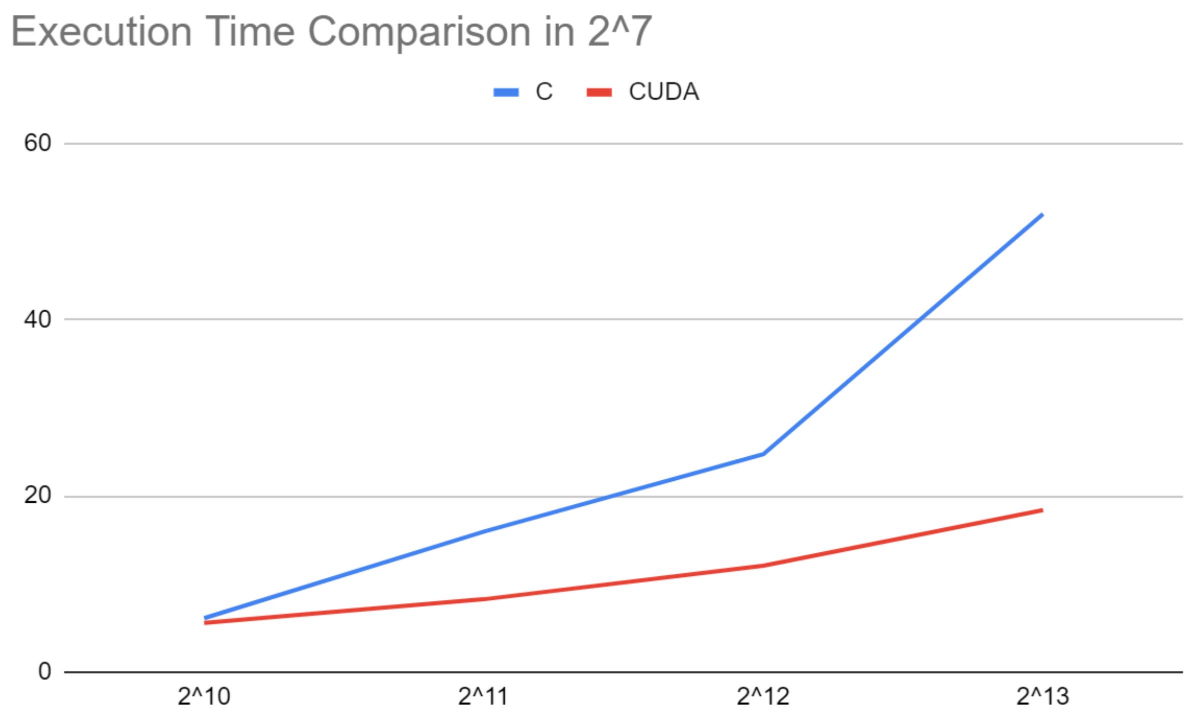


```
==3383== NVPROF is profiling process 3383, command: ./smithwaterman

Index start at 113 68
VRIPSA-RDYKXPEFVNFWFK-VAHEWAGQZYXIKHTYAIF-YB--QIYYCJGFNVANV-GKVETLAGKDTDFHMD
JXIAYIAZBYZCKKET-W-NTEIDNAWEG--Y------F--F-PLCSLZWLMF-B---LLH--PTTG-KHPBDHJB
==3383== Profiling application: ./smithwaterman
==3383== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.39%  55.750ms     11530  4.8350us  2.4630us  15.935us  mana(char*, int, char*, int, int*, bool*, int, unsigned long)
                    0.61%  339.77us         5  67.954us     864ns  321.50us  [CUDA memcpy HtoD]
      API calls:   69.69%  235.03ms         1  235.03ms  235.03ms  235.03ms  cudaMemcpyToSymbol
                   23.68%  79.868ms     11530  6.9260us  4.7330us  2.0645ms  cudaLaunchKernel
                    6.04%  20.372ms         4  5.0929ms  6.4930us  20.334ms  cudaMallocManaged
                    0.23%  780.57us         5  156.11us  18.587us  525.29us  cudaMemPrefetchAsync
                    0.13%  452.22us         4  113.06us  22.080us  372.79us  cudaMemcpy
                    0.11%  361.13us         4  90.282us  14.317us  285.81us  cudaFree
                    0.07%  219.53us       114  1.9250us     183ns  85.671us  cuDeviceGetAttribute
                    0.04%  138.11us         2  69.055us  37.327us  100.78us  cudaMemAdvise
                    0.00%  13.104us         1  13.104us  13.104us  13.104us  cuDeviceGetName
                    0.00%  8.6520us         1  8.6520us  8.6520us  8.6520us  cudaDeviceSynchronize
                    0.00%  8.0780us         1  8.0780us  8.0780us  8.0780us  cuDeviceGetPCIBusId
                    0.00%  5.7380us         1  5.7380us  5.7380us  5.7380us  cuDeviceTotalMem
                    0.00%  2.7270us         1  2.7270us  2.7270us  2.7270us  cudaGetDevice
                    0.00%  2.1860us         3     728ns     346ns  1.4840us  cuDeviceGetCount
                    0.00%  1.4830us         2     741ns     266ns  1.2170us  cuDeviceGet
                    0.00%     492ns         1     492ns     492ns     492ns  cuModuleGetLoadingMode
                    0.00%     456ns         1     456ns     456ns     456ns  cuDeviceGetUuid

==3383== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  4.0000KB  4.0000KB  4.0000KB  8.000000KB  6.080000us  Host To Device
       1  520.00KB  520.00KB  520.00KB  520.0000KB  42.17500us  Device To Host
       1         -         -         -           -  313.2120us  Gpu page fault groups
```

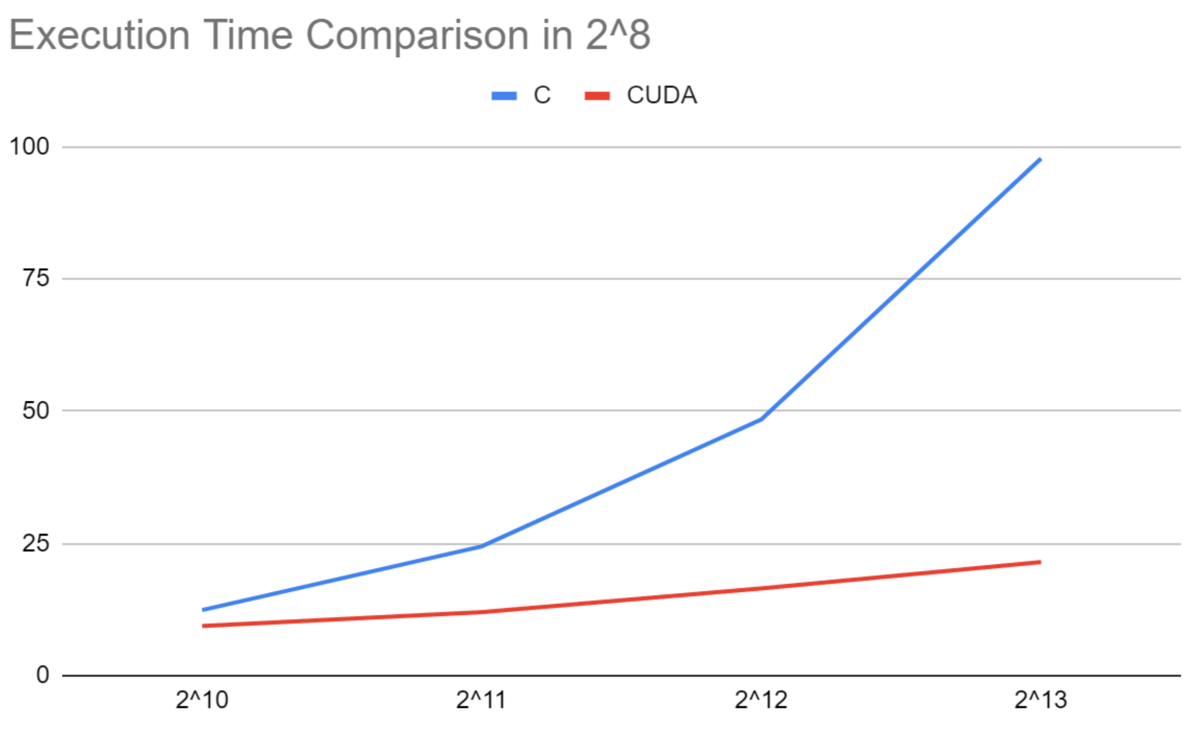
Sample CUDA Output

Input Sequence Array Sizes	**Sequential C**			**CUDA in C**		
	2^7	**2^8**	**2^9**	**2^7**	**2^8**	**2^9**
2^10	6.1216 ms	12.3885 ms	25.1443 ms	5.575 ms	9.3879 ms	19.277 ms
2^11	15.9405 ms	24.4397 ms	49.4683 ms	8.2695 ms	12.006 ms	23.656 ms
2^12	24.7330 ms	48.476 ms	99.2658 ms	12.067 ms	16.494 ms	23.183 ms
2^13	51.9746 ms	97.7898 ms	236.9554 ms	18.378 ms	21.473 ms	26.761 ms

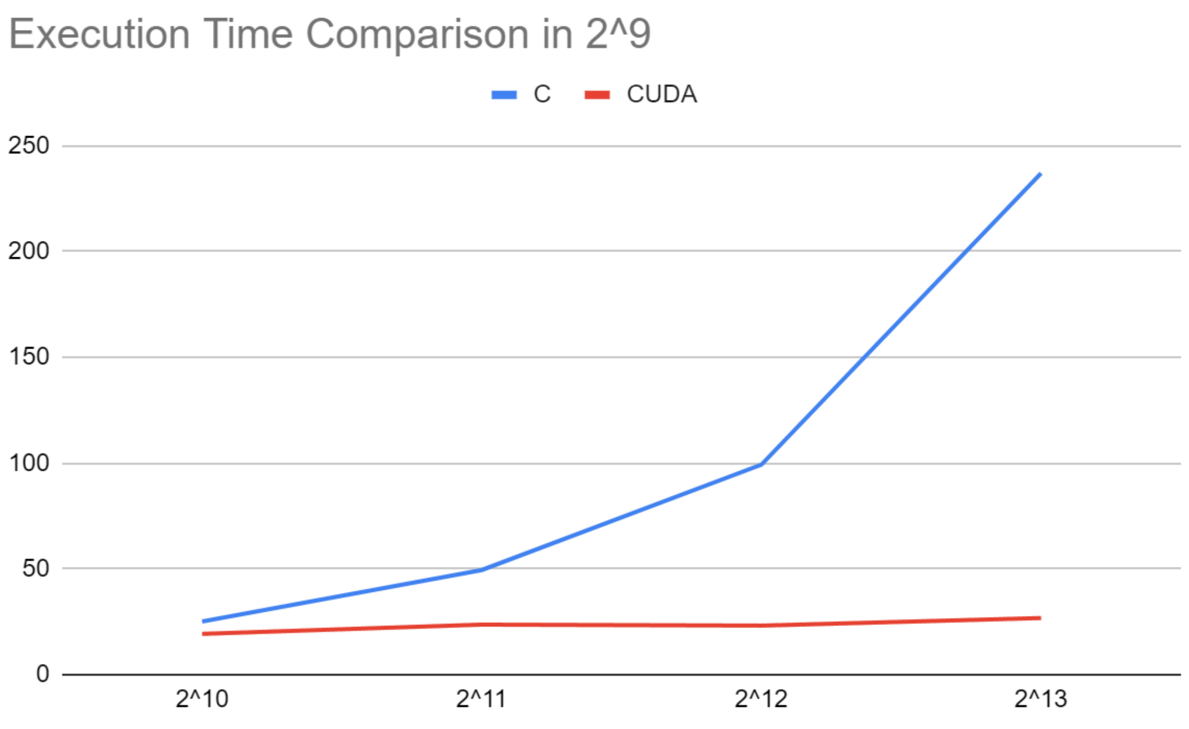
Execution times table



Execution time Comparison in 2^7



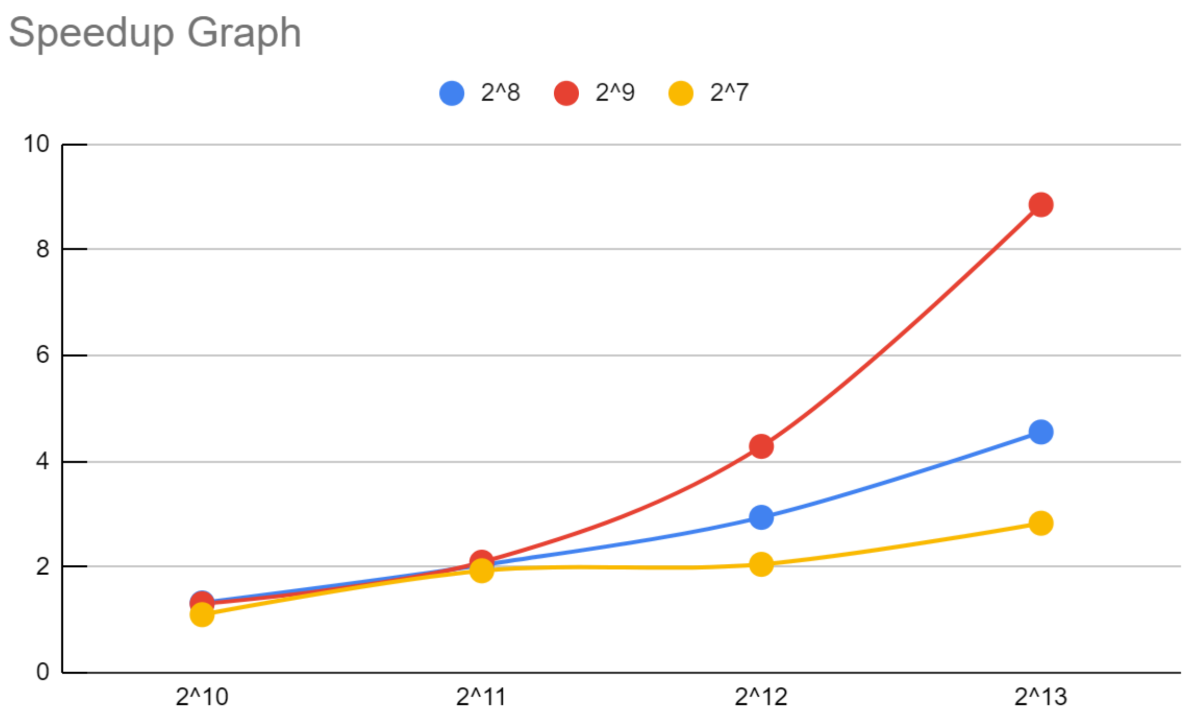
Execution time Comparison in 2^8



Execution time Comparison in 2^9

Input Sequence Array Sizes	**2^7**	**2^8**	**2^9**
2^10	1.098x	1.320x	1.304x
2^11	1.928x	2.036x	2.091x
2^12	2.050x	2.939x	4.282x
2^13	2.828x	4.554x	8.855x

Speedups table



Speedups graph

4. Discussion

In the sequential C implementation, there is a significant increase in execution time as the array size doubles. This growth is roughly exponential, which is typical for complex algorithms like sequence alignment. For example, with an input size of 2^10 x 2^7, the sequential implementation takes 6.1216 ms, and this increases to 236.9554 ms for an input size of 2^13 x 2^9.

The CUDA implementation also shows an increase in execution time with larger input sizes, but the rate of increase is less steep compared to the sequential approach. This indicates that the CUDA implementation handles larger inputs more efficiently than the sequential implementation. For instance, the execution time for an input size of 2^10 x 2^7 is 5.575 ms, increasing to only 26.761 ms for an input size of 2^13 x 2^9.

. As expected, both implementations' execution times rise with input size. However, the CUDA version consistently outperforms the sequential C code across all input sizes. The efficiency gain of CUDA becomes more prominent as the input size increases, with the CUDA implementation being approximately 8.855x faster than the sequential one at 2^13 x 2^9. The sequential implementation's exponential growth in execution time becomes impractical for very large input sizes, while the CUDA implementation scales better, making it more suitable for high-performance computing tasks involving large datasets like sequence alignment.

5. Conclusion

Overall, the execution times of the sequential C implementation are slower than that of the CUDA in C implementations. Our data also shows that there is a high variance in the speedups provided by the CUDA implementation. From our testing, this speedup ranges anywhere from 1.098x to 8.855x. The graphs also show a general upward trend that is certainly faster than linear and may even approach polynomial time, however, additional testing is necessary to further analyze these trends. Our results, however, show a large potential for the usage of CUDA in parallelizing and optimizing algorithms in the field of bioinformatics, especially for larger datasets.

6. References

Rocha, M., & Ferreira, P. G. (2018). Pairwise Sequence Alignment. Elsevier EBooks, 133–162. https://doi.org/10.1016/b978-0-12-812520-5.00006-7

Rudnicki, W. R., Jankowski, A., Modzelewski, A., Piotrowski, A., & Zadrożny, A. (2009). The new SIMD Implementation of the Smith-Waterman Algorithm on Cell Microprocessor. Fundamenta Informaticae, 96(1-2), 181–194. https://doi.org/10.3233/fi-2009-173

Smith, T. F., & Waterman, M. S. (1981). Identification of common molecular subsequences. Journal of Molecular Biology/Journal of Molecular Biology, 147(1), 195–197. https://doi.org/10.1016/0022-2836(81)90087-5

Storaasli, O., Strenski, D., & Inc, C. (n.d.). `Accelerating Genome Sequencing 100X with FPGAs Figure 2. Virtex-II Pro 50 FPGA speedup Figure 3. Virtex-4 LX160 speedup Cray XD1 (Virtex2) Speedup.