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,

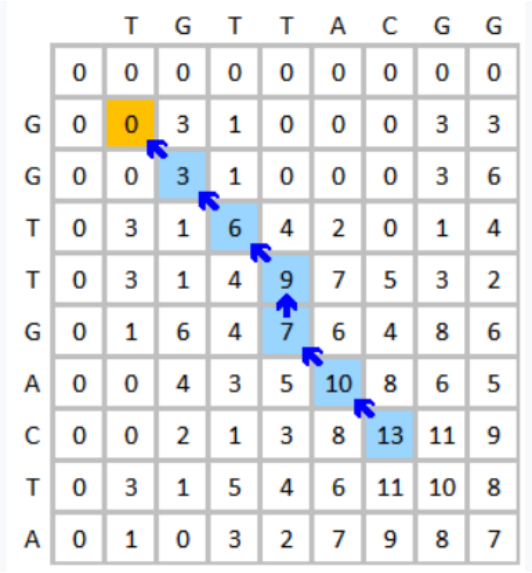
$H_{i,j-l} - W_l$ is the score if b_j is at the end of a gap of length l,

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 is a sample output, a table of the average runtime of the Sequential and CUDA implementation for different input array sizes, and a table containing speedup comparisons.

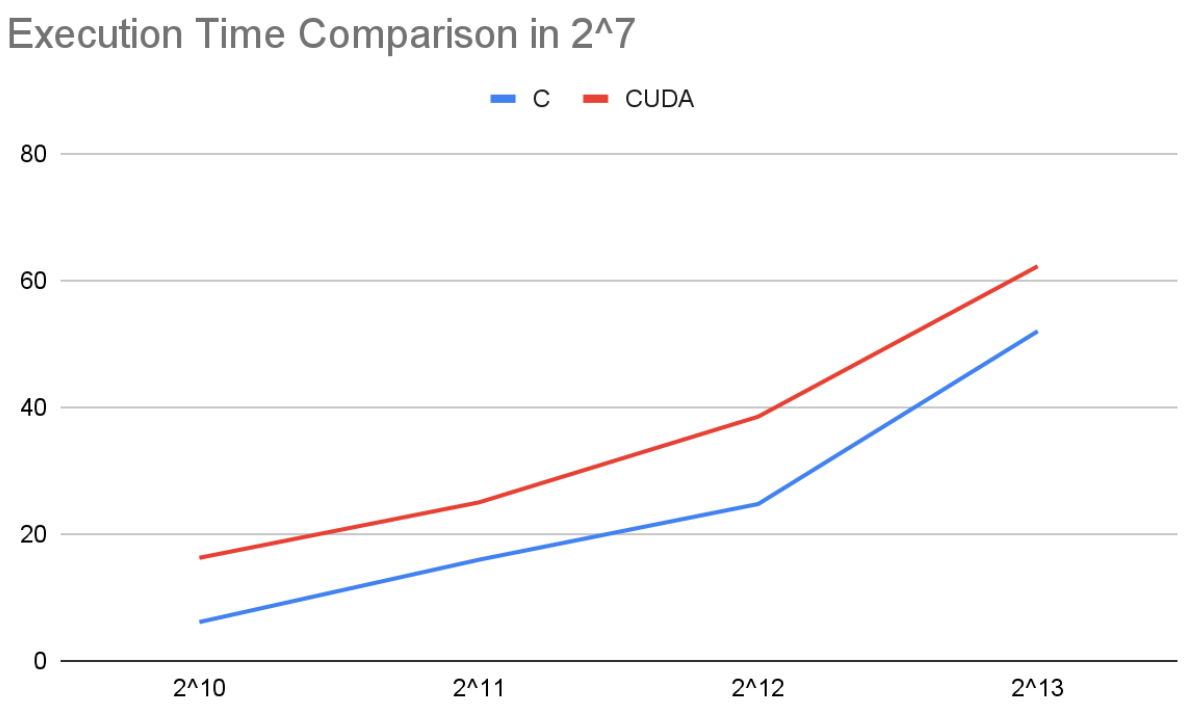
```
Index start at 94 103
PZIKXDZKWI--ZYM--XYVQEBHXVPSCBZYGHTZENRJQXH-S---I-ERFJVADRBKAHWHCFZZMEPNCH--DREDNFCHPW-----EN-----L-RDEMBE-QBLL-LHPH-S--W
PXIJE-TNWLPIRFVWMBYNQE---GKT-NWYH-LDZAV-AJWHSKYZMVDAF--SBGIRQH----QZKZ---HKHDR-----HPWJHRCVB-NXGQXMMKK-JFWE--LGI-HGMHLQHW
Maximum value in the scoring matrix: 78
Total kernel time (ms): 60.013000
==22875== NVPROF is profiling process 22875, command: ./smithwaterman
Average kernel time (ms):6.001300
Index start at 94 103
PZIKXDZKWI--ZYM--XYVQEBHXVPSCBZYGHTZENRJQXH-S---I-ERFJVADRBKAHWHCFZZMEPNCH--DREDNFCHPW-----EN-----L-RDEMBE-QBLL-LHPH-S--W
PXIJE-TNWLPIRFVWMBYNQE---GKT-NWYH-LDZAV-AJWHSKYZMVDAF--SBGIRQH----QZKZ---HKHDR-----HPWJHRCVB-NXGQXMMKK-JFWE--LGI-HGMHLQHW
Maximum value in the scoring matrix: 78
Success. Traceback string outputs are identical.
==22875== Profiling application: ./smithwaterman
==22875== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.79%  163.31ms     11530  14.164us  3.0400us  16.703us  p_mana(char*, int, char*, int, int*, bool*, int, unsigned long)
                    0.21%  347.23us         5  69.445us     800ns  329.88us  [CUDA memcpy HtoD]
      API calls:   53.64%  157.42ms     11530  13.653us  3.0100us  842.65us  cudaLaunchKernel
                   33.45%  98.160ms         1  98.160ms  98.160ms  98.160ms  cudaMemcpyToSymbol
                    6.94%  20.374ms         4  5.0935ms  6.0120us  20.340ms  cudaMallocManaged
                    5.37%  15.758ms         1  15.758ms  15.758ms  15.758ms  cudaDeviceSynchronize
                    0.27%  790.66us         5  158.13us  16.689us  448.79us  cudaMemPrefetchAsync
                    0.15%  449.71us         4  112.43us  17.705us  371.25us  cudaMemcpy
                    0.10%  284.33us         4  71.081us  10.182us  218.64us  cudaFree
                    0.04%  128.18us       114  1.1240us     136ns  49.640us  cuDeviceGetAttribute
                    0.03%  79.532us         2  39.766us  19.348us  60.184us  cudaMemAdvise
                    0.00%  11.570us         1  11.570us  11.570us  11.570us  cuDeviceGetName
                    0.00%  5.7260us         1  5.7260us  5.7260us  5.7260us  cuDeviceGetPCIBusId
                    0.00%  5.0680us         1  5.0680us  5.0680us  5.0680us  cuDeviceTotalMem
                    0.00%  2.1760us         1  2.1760us  2.1760us  2.1760us  cudaGetDevice
                    0.00%  1.5160us         3     505ns     178ns     974ns  cuDeviceGetCount
                    0.00%  1.2030us         2     601ns     262ns     941ns  cuDeviceGet
                    0.00%     544ns         1     544ns     544ns     544ns  cuModuleGetLoadingMode
                    0.00%     242ns         1     242ns     242ns     242ns  cuDeviceGetUuid

==22875== 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  5.696000us  Host To Device
       1  520.00KB  520.00KB  520.00KB  520.0000KB  41.72700us  Device To Host
       1         -         -         -           -  323.0330us  Gpu page fault groups
```

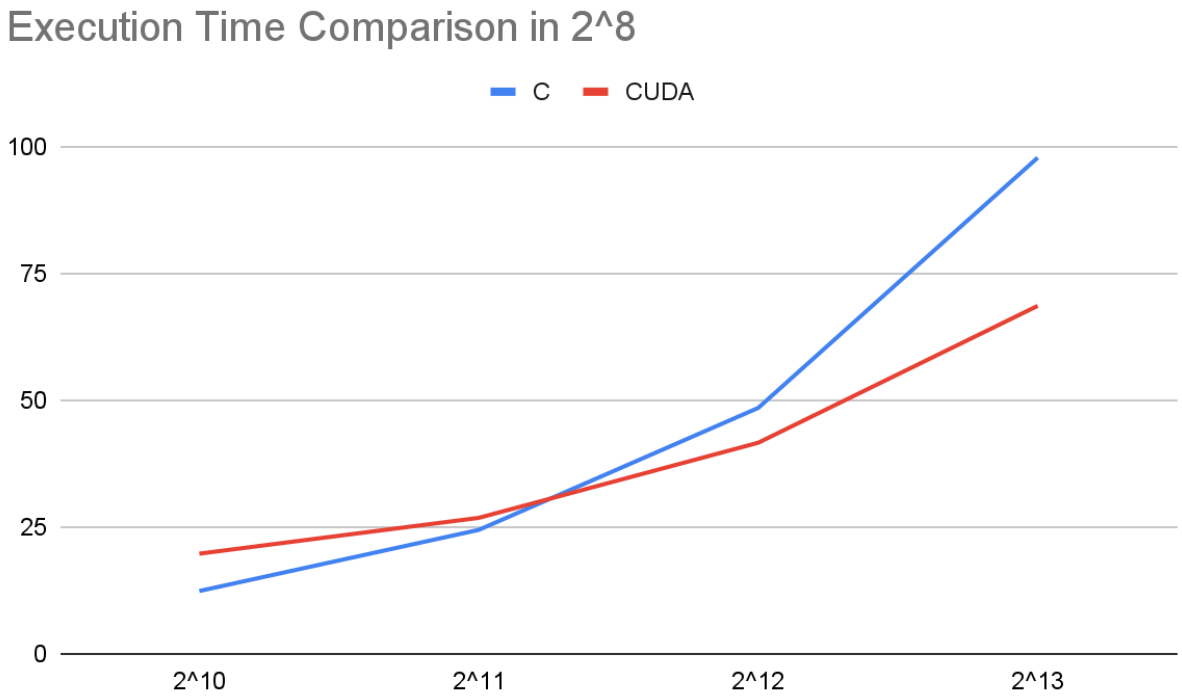
Sample Output

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	16.251 ms	19.751 ms	26.034 ms
2^11	15.9405 ms	24.4397 ms	49.4683 ms	24.982 ms	26.808 ms	33.735 ms
2^12	24.7330 ms	48.476 ms	99.2658 ms	38.506 ms	41.611 ms	53.526 ms
2^13	51.9746 ms	97.7898 ms	236.9554 ms	62.216 ms	68.589 ms	89.936 ms

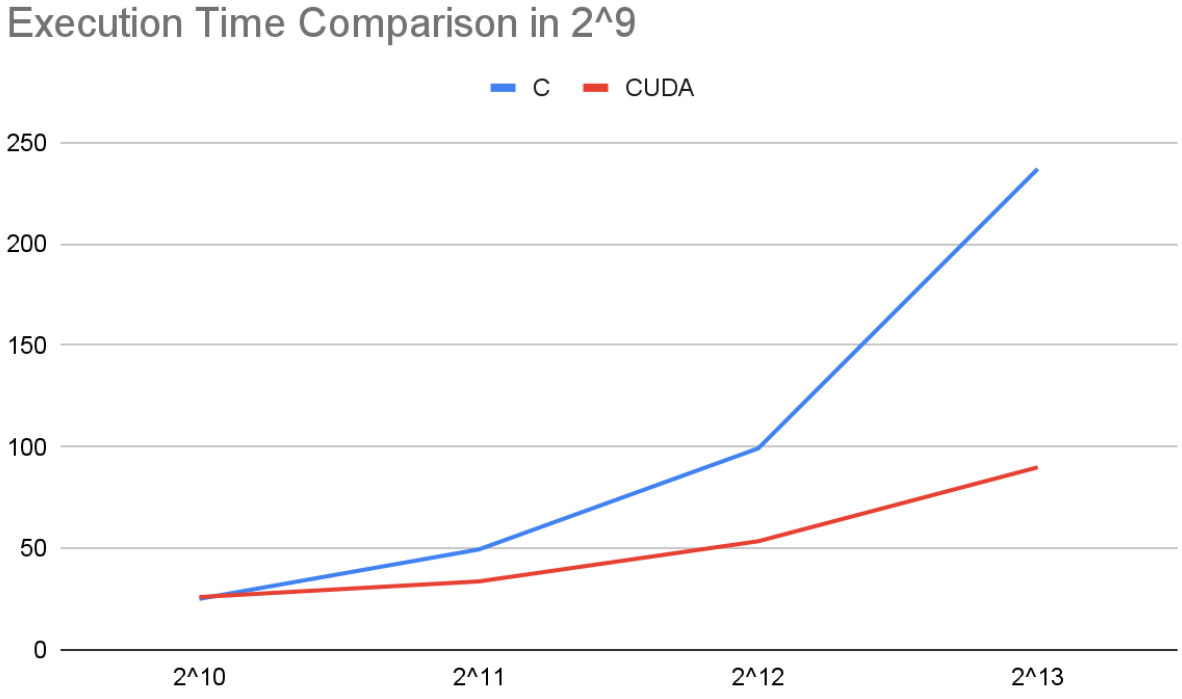
Execution times table



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	*2.6546x slower*	*1.5943x slower*	*1.0354x slower*
2^11	*1.5672x slower*	*1.0969x slower*	1.4664x faster
2^12	*1.5568x slower*	1.1650x faster	1.8545x faster
2^13	*1.1970x slower*	1.4257x faster	2.6347x faster

Speedups table

4. Discussion

As depicted in the execution times table, the sequential implementation generally performs better or comparably for smaller arrays. However, as the array size increases, the CUDA implementation starts to outperform the sequential one, with the most significant improvements observed for the largest array sizes tested (2^8 by 2^12 and above). This trend highlights the efficiency of parallel processing in CUDA, particularly for larger datasets.

The speedup table further emphasizes this point, illustrating how the speedup improves with increasing array size. For example, the speedup for 2^9 by 2^11 approaches 1.5x, and the speedup for 2^9 by 2^13 reaches over 2.6x for the largest array size tested. In contrast, the smallest array size (2^7 by 2^10) shows no improvement, and even results to a slowdown of about 2.65x. This indicates that the overhead associated with CUDA initialization and data transfer becomes more prevalent and noticable at smaller data sizes, and is more justified for larger workloads, where CUDA's parallel processing can be fully leveraged.

Additionally, the data clearly demonstrates the scalability and efficiency gains of the CUDA implementation for computationally intensive tasks involving large datasets. While the sequential implementation remains to be better for smaller arrays, the benefits of CUDA become increasingly evident as the array size grows, making it a valuable approach for handling large-scale computational problems such as sequence alignment.

Given the platform limitations that the proponents used (Google Colab), the array sizes that were tested could not go beyond a combined size array of 2^23. Nevertheless, the implemented program when put in a different environment with lesser constraints has the potential to provide large amounts of speedup for large datasets.

.

5. Conclusion

Overall, the execution times of the sequential C implementation are faster than that of the CUDA in C implementation when the input array sizes are small. This behavior quickly changes as the input array sizes increase, and shows a trend of increasing speedups as data sizes are increased. From our testing, this speedup ranges from being 2.655x slower to 2.635x faster as array sizes increase. The graphs also show that while smaller array sizes lead to slowdown, larger data sizes shows a trend of increasing speedps for large array sizes, 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.