Protein sequence alignment with GPU: Database Optimization

Ahmad Faiz Bin Mohd Rahi
Faculty of Electrical Engineering
Universiti Teknologi MARA – (UiTM)
40000 Shah Alam, Selangor, Malaysia
faiztm90@gmail.com

***Abstract*—This paper present a protein sequence alignment accelerated with Graphics Processing Units (GPUs). In bioinformatics, alignments are commonly performed in genome and protein sequence analysis for gene identification and evolutionary similarities. For such analysis, there are a few approaches, every single different in accuracy and computational difficulty. Smith-waterman (SW) considered as the greatest algorithm for its accuracy in same scoring. In the other hand, it is not suitable to be used by life scientists as it is experience executions time on general purposed. Through this paper we focus on Smith-Waterman to discover the construction features of Graphic Processing Units (GPUs) and determine the difficulties in the hardware construction as well as the software improvements needed to put on the program construction on the GPU. In comparison with the state-of-the-art implementation on an NVIDIA Geforce 610M graphics card, our implementation reports a 1.9 times performance improvement in terms of execution time.**

***Index Terms*—Graphic Processing Unit** *(GPU),* **Smith-Waterman Algorithm** *(SW),* **Compute Device Unified Architecture** *(CUDA),* **Protein Sequence Alignment**

I. INTRODUCTION

In the era of advance technology, people desire new and very powerful chip in their graphic hardware. Graphic processing unit are called as GPUs is increasing rapidly in the last few years. The usually used of graphic processing unit (GPUs) are to accelerate the graphic rendering. There several rule for graphic processing unit (GPUs) such as it is massive parallel. It means the computations can be broken down into hundreds or thousands of independent units of work. The best performance when all of the cores are kept busy, exploiting the inherent parallel nature of the graphic processing unit (GPUs). Seemingly simple, vectored MATLAB calculations on arrays with hundreds of thousands of elements often can fit into this category. Second, it is computational intensive. It means the time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory. Because a GPU is attached to the host CPU via the PCI Express bus, the memory access is slower than with a traditional CPU. This means that your overall computational speedup is limited by the amount of data transfer that occurs in your algorithm.

Smith-Waterman (S-W) algorithm is an optimal sequence alignment method for biological databases, but its computational complexity makes it too slow for practical purposes. Heuristics based approximate methods like Fasta and Blast provide faster solutions but at the cost of reduced accuracy. Also, the expanding volume and varying lengths of sequences necessitate performance efficient restructuring of these databases. Thus to come up with an accurate and fast solution, it is highly desired to speed up the S-W algorithm. Result is an improved performance which is better than the fastest available GPU implementation.

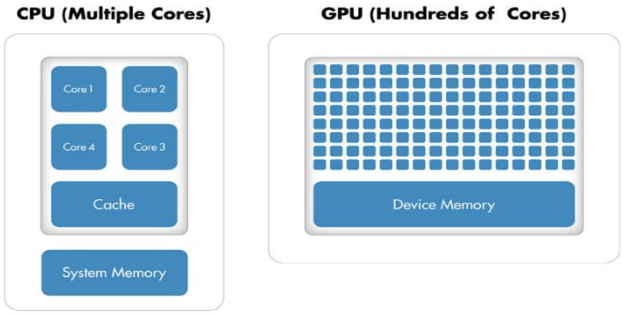

Fig 1: Comparison the number of cores of CPU type and GPU

At first, before graphic processing unit (GPUs) are used in computing, we are used Multicore machines and hyper-threading technology. These have enabled scientists, engineers, and financial analysts to speed up computationally intensive applications in a numerous of disciplines. Today, next type of hardware gives even higher computational performance, which is the graphics processing unit (GPUs).

The SW algorithm, besides being the most sensitive for searching protein databases for sequence similarities, is also the most time consuming [2, 3, 4]. A protein database is a database containing protein sequences with known functionality. SW provides a score of similarity between two sequences [5]. This similarity score is sometimes referred to as the SW score.

Instead of looking at an entire sequence at once, the S-W algorithm compares multi-lengthed segments, looking for whichever segment maximizes the scoring measure. The algorithm itself is recursive in nature [100]:

$$H_{ij} = \max \begin{cases} H_{i-1, j-1} + s(a_i, b_j); \\ H_{i-k, j} - W_k; \\ H_{i, j-1} - W_1; \\ 0; \end{cases} \quad (1)$$

Almost all desktops and servers currently manufactured have GPUs in addition to having multiple CPU cores. The presence of heterogeneous cores (CPU cores and GPU cores) on these

machines further complicates the task of programming them. Several frameworks have been designed to allow developers to write programs for these heterogeneous machines. The following section discusses some of these.

NVIDIA CUDA is an extension of the C++ programming language. It extends the base C++ language to provide data parallel constructs to write code that is to run on the GPU[6]. Programs written in CUDA have two distinct parts – the part of the program written in base C++ which is to execute on the host processor and code written using the CUDA language extensions that is to execute on the device (GPU). The CUDA toolkit provides a compiler, nvcc, to compile CUDA programs into executable. When the resulting executable is run, it starts executing on the CPU and then initiates required DMA transfers and GPU kernel executions.

From the figure 2, it show the CUDA hierarchy of threads block and grids. From our PC we can use CUDA software to program the GPU. There are multiprocessor in the GPU that will execute several of threads block. One multiprocessor can be run by multiple threads block, or in parallel threads block using threads switching. CUDA cores are the processing elements within a multiprocessor in group of 32 called warps

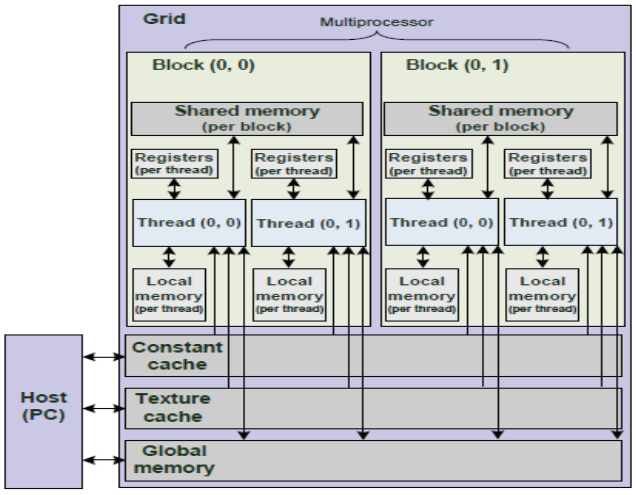

Fig 2: Programming model. CUDA hierarchy of threads, blocks and grids.[5]

On the GPU, a hierarchy of memory architecture is available for the programmer to utilize. As provided by the CUDA programming guide, these include Registers: Read-Write per-thread

- Local Memory: Read-write per –thread
- Shared Memory: Read-write per-block
- Global Memory: Read-write per-grid
- Constant Memory: Read-only per-grid
- Texture Memory: Read-only per-grid

Figure 3 show that CUDA memory hierarchy that consist of host (PC), global memory, texture cache, constant cache, and block where the block consists several parts. Global memory in the figure 2 is the GPUs RAM. Then the constants cache which is a read-only portion of global memory. It caches at each multiprocessor and accessing it is as fast as accessing a register [5]. The other types of memories are shared memory and local memory, where shared memory is a fast memory used for inter-thread communication within a thread block and local memory is a per thread portion of the global memory used for function calls and register spills. Additionally, each multiprocessor offers a bank of registers, shared between its processors [5]

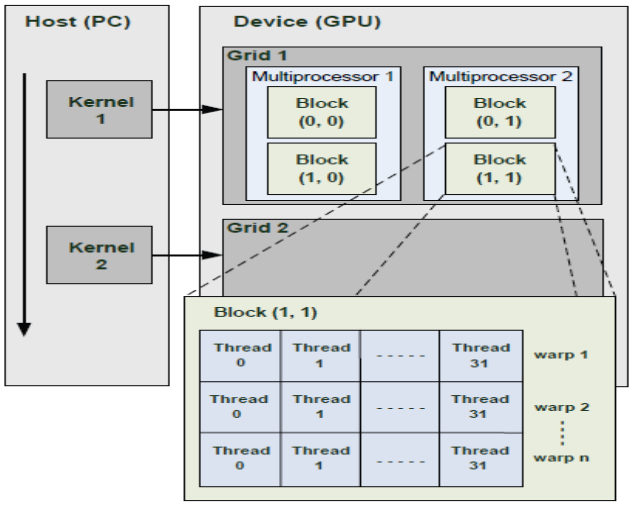

Fig 3: Memory model. CUDA memory hierarchy.[5]

II. METHODOLOGY

A. General design

NVIDIA Compute Unified Device Architecture (CUDA) [12] is implement for the GPU programming (device code) in combination with C++ for the PC programming (host code) as CUDA exists as the advance GPU programming currently similar to other present GPU applications protein arrangement from the Swiss Prot database [6] are counted for arrangement because the structure of protein arrangement is complicated compared to DNA. The explanations of the application is described in figure 4. The host code regularly taking into consideration with loading data structures, copying them to GPU and copying back, at the showing the outcome. The query arrangement, improved database and other data are copied to the GPU. SW algorithm is used to blast off the device code, to align with the database sequence. Throughout, the number of such arrangement are too small as about 20 top scoring arrangement are returned, where about more than 500,000 from Swiss Prot database.

Every processing component in our application is considered to self-generate a full alignment between a queries arrangement and database arrangement. Efficient resource utilization is the sequence of the exclusion of the need for inter processor communication. It is conservable to keep all processor well filled as the GPU used for application contain 48 processors cores, while the newest Swiss Prot has more than 500,000 arrangement [7].

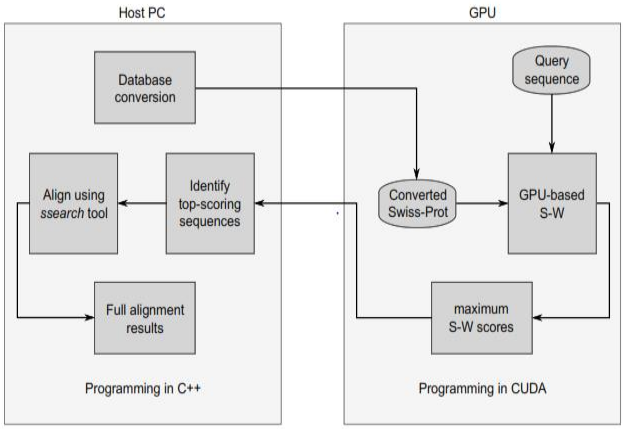

Fig 4: Description of GPU implement.

B. Database Organization

Names and other biological info are the part of sequences description of the Swiss Prot database which being monitored through Fasta Format. In addition, the GPU implementation turns them to a standard GPU format which suits the ability of the device [5]. The database is also directly converted into Fasta Format and saved in the new format. It just required to be converted once. Figure 5 shows the conversion process.

1) *Arrangement:* A thread is a basic element of the CUDA programming model which perform an example of the code. Asset of thread which perform a CUDA programming in parallel; where it has admission to registers and per thread memory [8]. Warps is the GPU processors which perform threads in group of 3D operation on GT200-class. GPU's can be enhanced a great deal by having threads in half-warp (16 threads) performed the similar code path and access memory in a hear vicinity threads of half warp cannot perform their task, individually as it has to wait for each other to finish. Database sequences are arranged by length to reduce length dissimilarities between nearby threads as shown in figure 5(b) which may minimize the waiting time. Description of the sequence are located in isolated file which is being not uploaded to the GPU, which conserve memory and reduce load time. In addition, sequence character are change with numeric indexes to assist simpler substitution matrix lookup.

2) *Concatenation:* Groups of 16 sequences are processed after being taken in sequence sets which considering half-warp of threads functioning for them such showed in Figure 5(c), after arrangements. Numerous sequence sets still have various sizes, even though through arrangement by length has somewhat balanced task inside a sequence set are concatenated with discarded sequences to shape sequence group. The largest sequence in the sets are almost like same with the total length of each sequence groups. This cause in equivalent task for each thread in a half-warp processing a sequence set. In between the concatenated sequences, sequence terminators are placed; which may inform GPU kernel to perform new arrangement. The threads will wait for the other threads in the half-warp to perform accomplishment as the terminator are implant at each end of the sequence group, indicating the end of group concatenated.

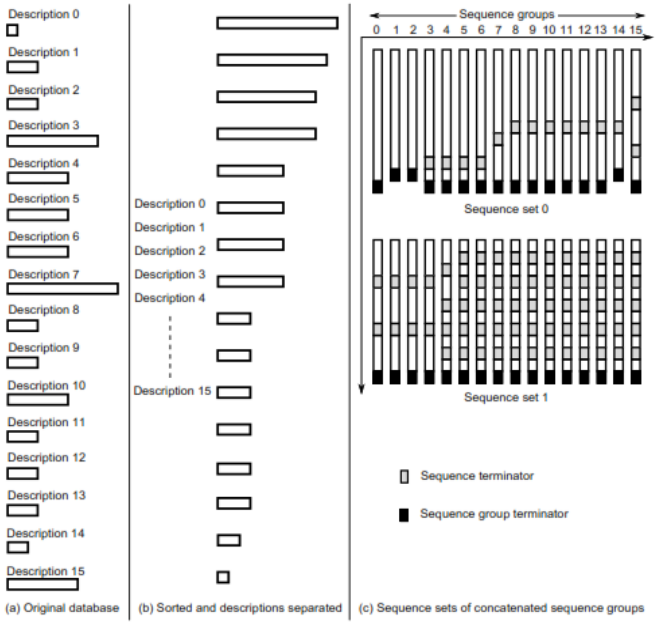

Fig 5: The Database conversion process.

3) *Interlacing:* As all the database arrangement have been processed into 16-wide sets of arrangement groups they are marked to five. The arrangement sets are marked in an interlinking subsets contains of 8 characters from every arrangement group.

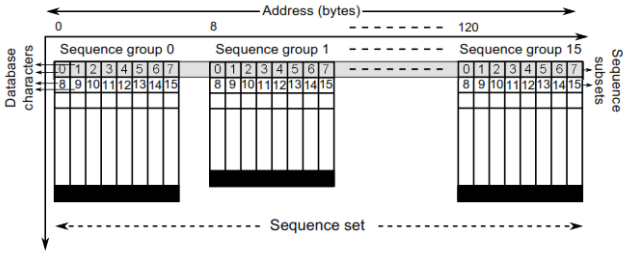

Fig 6: Sequence storing as interlaced subsets

C. Temporary data reads and writes

While developing the GPU implementation, the memory bandwidth represented a serious bottleneck. A number of steps have been taken to optimize for high performance by reducing the number of memory accesses, the frequent temporary data accesses in particular. As no traceback is performed on the GPU, S-W matrix values do not need to be saved for the entire execution time and can be overwritten. As such, only a single column of S-W scores is kept. This score column stores values to the left of the currently processing column, i.e. $H_{i-1,1 \le j \le N}$ in Equation 1. The size of this temporary data column is set based on the size of the query sequence, not the database sequence, so that the column can have one fixed size for all database sequences. This usually requires less memory, as it is unlikely that the query sequence will be as long as the longest database sequence. The temporary data column is set to zero whenever a

new database sequence is started. In addition to this temporary score column, variables are used to keep the values of the upper and upper-left cells required by the algorithm, i.e. $H_{i,j-1}$ and $H_{i-1,j-1}$ in Equation 1. To support affine gap penalties, another temporary data column is added for D values. Additionally, an upper E value is kept (see Equation 1).

Each S-W iteration involves reading and writing two temporary values (score and D), for four accesses in total. When both are non-coalesced, 32 byte reads/writes are issued for each access. This means that per half-warp 16 threads × 32 bytes × 2 values × 2 read/write = 2048 bytes of bandwidth is used, resulting in a major memory bottleneck. The optimization steps mentioned below decrease this to one 128-byte coalesced read and write for every second iteration. This is a 16 times bandwidth improvement and requires only 1 instead of 64 accesses. 128 bytes is the largest allowed coalesced access size, and is faster than multiple smaller coalesced accesses. The optimizations are as follows:

- Smaller, 16-bit data type for the temporary values, cutting the theoretically required bandwidth in half and allowing for better coalescing.
- Each thread stores only one data value in turn, resulting in an interlaced storage scheme. Instead of direct array accesses, a pointer into the temporary storage is started at the thread *id*, and increased by the total number of threads to move to the next element of the S-W H matrix. Each thread in a half-warp then reads a 2-byte coalesced value, meaning that instead of two 32-byte accesses per thread, two such accesses take place per half-warp. This sixteen times bandwidth improvement results in an almost ten times net speedup.
- To halve the number of memory accesses, the temporary score and D values are interlaced. This is done by defining a data structure consisting of these values and using it to access the score and D values for an iteration in one go. At this point, a thread accesses two 2-byte values in one read, for a total of 16 ×2 ×2 bytes bandwidth per half warp. The result is a 64-byte coalesced access.
- Finally, two temporary values are interlaced to move to 128-byte accesses. This has an additional benefit of temporary reads/writes only being required for every second query sequence symbol processed.

D. Substitution matrix accesses

To align the proteins, we need to use a substitution matrix, which is accessed every time two symbols are aligned, making its access time critical to the implementation's performance. Substitution matrix (e.g. BLOSUM 62) accesses are random and are completely dependent on the database sequence, complicating the choice of memory used. GPU's Global memory is not a good choice for such a frequent usage due to its high access time. Also the random nature of substitution matrix accesses makes coalescing very difficult. As an alternative, the substitution matrix is stored in texture memory. Texture memory is a cached window into global memory that offers lower latency and does not require coalescing for best performance. It is thus well suited for random access. Texture memory has the ability to fetch four values at a time. This mechanism can be used to fetch four substitution matrix values from a query profile. A query profile is shown in Figure 7. It is a type of substitution matrix where, instead of the protein alphabet, the query sequence is used along the top row. This means that for a given database character, the substitution matrix is not random anymore: multiple substitution scores can be loaded simultaneously when aligning the query with a database character. Furthermore, query sequence lookups are not required anymore; only the current position within the query is needed to index into the profile. A query profile is generated once for every query sequence. Each query profile column stores values for 23 characters. The number of columns and hence the memory requirement for a query profile depends on the length of the query sequence. The GeForce 6100M GPU used for our implementation has 16KB of texture cache per multiprocessor. This means that a query sequence having more than ⌊16 × 1024/23⌋ = 712.35 characters will result in increased cache misses, as described in [14]. Tests were performed to quantify the texture cache miss rate, which was shown to be very low. For example, aligning an 8000 character query sequence resulted in 0.009% miss rate. For smaller sequences, the miss rate was reported to be even lower. Using this query profile method resulted in a 17% performance improvement with Swiss-Prot.

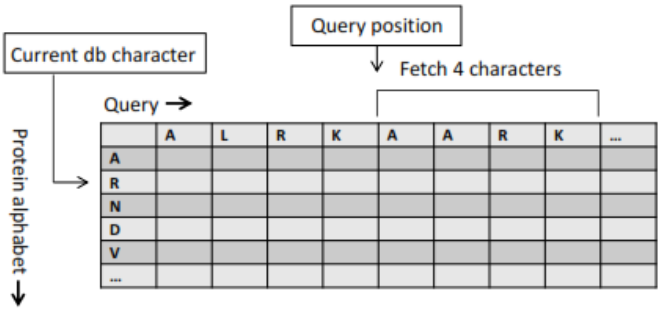

Fig 7: Query profile

III. Results And Discussions

We have tested graphic algorithm SW on an Asus A55V series running on Window 7 Premium. The computer specification as below:

- Intel Core i5 – 3210M, 2.5GHz
- 4 GB DDR3 1600 MHz SDRAM
- NVIDIA® GeForce® 610M with 2GB DDR3 VRAM
- Swiss-Prot release October 16, 2013
- CUDA toolkit version 3.1
- Substitution matrix BLOSUM 62
- Gap penalty: -10 and gap extend penalty: -2 (these do not influence the execution time though)

The amount of processor and RAM contain in our systems are irrelevant, this is because all SW calculations are performed strictly on the GPU and these are the calculations that we have

clocked. We have provided the computers configuration simply for an information purposes. It does not have an effect on how the GeForce 610M performs its calculations.

TABLE I: EXECUTION TIME OF BLAST, SSEARCH AND GASW FOR THE PROTEIN. RATIO IS BETWEEN EXECUTION TIME FOR SSEARCH AND GASW.

		Execution Time			
Query Sequence	Length	BLAST	SSEARCH	Gasw	Ratio
P02232	144	25.00	7.03	3.24	2.17
P05013	189	34.40	7.11	3.46	2.05
P14942	222	34.80	8.61	4.30	2.00
P07327	375	35.00	11.12	4.97	2.24
P01008	464	45.40	11.31	5.80	1.95
P03435	567	85.80	12.37	7.08	1.75
P27895	1000	106.10	16.14	11.07	1.46
P07756	1500	115.20	21.38	15.99	1.34
P04775	2005	116.30	25.01	20.60	1.21
P19096	2504	127.30	31.35	28.03	1.12
P28167	3005	165.20	33.60	28.70	1.17
P0C6B8	3564	196.90	37.42	34.01	1.10
P20930	4061	297.60	40.71	37.98	1.07
Q9UKN1	5478	524.40	60.17	51.02	1.18

From table 1, we can see the execution time for the Blast, Ssearch and Gasw program. From this result, we known that execution time for Blast program is slow running than other. This is because the Blast program compares protein sequences to sequence databases and calculates the statistical significance of matches. Blast can be used to infer functional and evolutionary relationships between sequences as well as help identify members of gene families [13]. Blast is the heuristics based and it is fast but it do not guaranty optimal alignment sequence.

For Ssearch, it also like as Blast program that was heuristics based and it is fast but do not guaranty optimal alignment sequence. Ssearch (SSE2) is an accelerated and multi-threaded version of ssearch, where ssearch is a CPU-based S-W alignment tool that can be found in the Fasta suite of applications [10]. The SSE2 optimizations, described in [14] utilize modern CPU's vector extensions for a performance increase. The result was showed Ssearch is faster than Blast program, but it is still slower than Gasw program.

From the result in table 1, it was observed that Gasw is better than Blast and Ssearch on running the protein sequence alignment. The execution times of Gasw is about 2 times faster than Ssearch when it running on 144 length of protein query sequence. However, after the length of protein reached about 1000 the gap difference of execution time between Ssearch and Gasw became narrowed. But, the execution time of Gasw still faster than Ssearch. We can see it on figure 9.

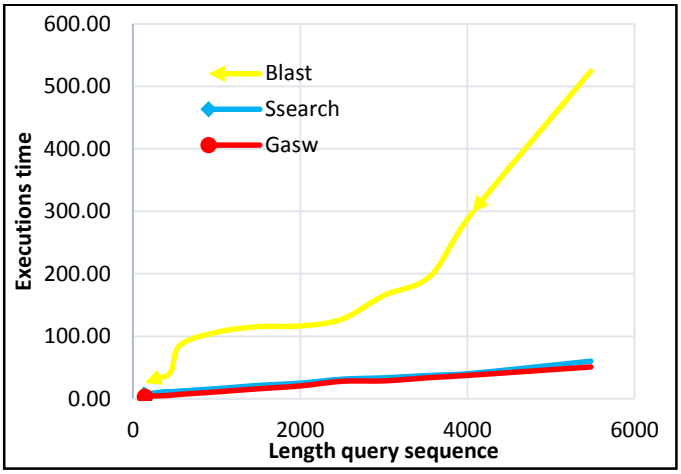

Fig 8: Graph for execution time for Gasw, Ssearch and Blast.

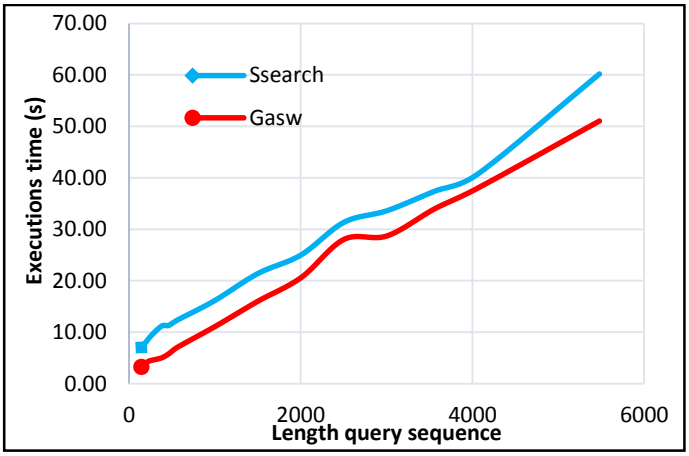

Fig 9: Graph for execution time between Gasw and Ssearch

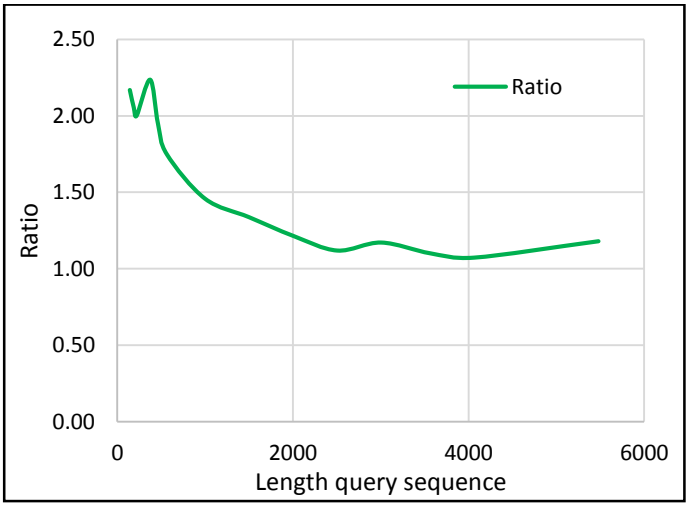

Fig 10: Graph for ratio between Gasw and Ssearch

From figure 10, the ratio shows that Gasw program that had been implement on GPU is faster about 1.9 faster than Ssearch at the length sequence less that 1000 sequence. After length 1000

sequence the ratio goes down from 1.5 to 1.1. From this result we have found that the GPU architecture has issues that can have a negative impact on the execution of certain algorithms. A single multiprocessor contains 16KB of shared memory and 8,192 registers. Let's say, for example, the kernel is running 768 threads per multi-processor. Then there will only be roughly 21 bytes of shared memory and 10 registers available per thread, assuming each thread's data is not dependent on other threads. For an algorithm such as Smith-Waterman that uses several hundred MB(s) of memory, this means that there must be several reads to the global memory in the kernel. Since all the cache memory has been used, the proccessor need to access the global memory. Thus, the execution time or clock cycle will be longer than usual. In this work, we took lower the miss rate of the cache memory that also can effect the execution time of the instruction.

IV. CONCLUSION

Through this work we have found that SW can be effectively mapped to the GPU architecture using CUDA. We achieve a 1.9 times faster over the serial method Ssearch. The new implementation improves the performance by reducing the number of memory accesses and optimizing the database organization. The database is organized in equal length sequence sets resulting in an equal workload distribution for all the threads of each multiprocessor on the GPU. In comparison with the state-of-the-art implementation on an NVIDIA Geforce 610M graphics card, our implementation reports a 1.9 times performance improvement in terms of execution time. For recommendation we need to use a better GPU to get faster and stable for execution purpose such as GPU that consists of large number of cores and memory.

ACKNOWLEDGEMENT

The author would like to thank Universiti Teknologi MARA (UiTM) Malaysia, Miss Ili Shairah Abdul Halim, Mr Syed Abdul Mutalib and Mr. Ahmad Syafiq Abd Suki for all guidance, support and advice provided to me throughout the final year project.

REFERENCES

[1] Mathworks Home Page. http://www.mathworks.com/
[2] Manavski, S.S., Valle, G.: Cuda compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 2008, 9(Suppl 2):S10 (2008)
[3] Sanchéz, F., Salamí E., Ramierez, A., Valero, M.: Performance Analysis of Sequence Alignment Applications. In: Proceedings of the IEEE International Symposium on Workload Characterization. October. 2006. pp 51-60. (2006)
[4] Walker, J.M.: The Proteomics Protocols Handbook, Humana Press, pp 504 (2005)
[5] Hasan et al.: DOPA: GPU-based protein alignment using database and memory access optimizations. BMC Research Notes 2011 4:261
[6] NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide. http://www.nvidia.com/object/cuda_develop.html, (2008)
[7] "http://www.uniprot.org", *Universal Protein Resource*, October 2010
[8] M.A. Kentie, "Biological Sequence Alignment Using Graphics Processing Units", *M.Sc. Thesis CE-MS-2010-35*, Computer Engineering Laboratory, Technical University Delft, The Netherlands, 2010.
[9] NVIDIA Corporation: http://www.nvidia.in/object/cuda-parallel-computing-in.html
[10] UVA: FASTA program webpage. http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml (2008)
[11] NCBI: National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt. May 2008. (2008)
[12] Fermi TM "NVIDIA's Next Generation CUDA TM Compute Architecture", White paper NVIDIA Corporation, 2009.
[13] BLAST program webpage : http://blast.ncbi.nlm.nih.gov/Blast.cgi
[14] M. Farrar, "Striped Smith-Waterman Speeds Database Searches Six Times over other SIMD Implementations", Bioinformatics, vol. 23(2), pages 156–161, 2007.