Sequence Alignment with GPU: Performance and Design Challenges

Gregory M. Striemer and Ali Akoglu
Department of Electrical and Computer Engineering
University of Arizona, 85721
Tucson, Arizona USA
{gmstrie, akoglu}@ece.arizona.edu

***Abstract*—In bioinformatics, alignments are commonly performed in genome and protein sequence analysis for gene identification and evolutionary similarities. There are several approaches for such analysis, each varying in accuracy and computational complexity. Smith-Waterman (SW) is by far the best algorithm for its accuracy in similarity scoring. However, execution time of this algorithm on general purpose processor based systems makes it impractical for use by life scientists. In this paper we take Smith-Waterman as a case study to explore the architectural features of Graphics Processing Units (GPUs) and evaluate the challenges the hardware architecture poses, as well as the software modifications needed to map the program architecture on to the GPU. We achieve a 23x speedup against the serial version of the SW algorithm. We further study the effect of memory organization and the instruction set architecture on GPU performance. For that purpose we analyze another implementation on an Intel Quad Core processor that makes use of Intel's SIMD based SSE2 architecture. We show that if reading blocks of 16 words at a time instead of 4 is allowed, and if 64KB of shared memory as opposed to 16KB is available to the programmer, GPU performance enhances significantly making it comparable to the SIMD based implementation. We quantify these observations to illustrate the need for studies on extending the instruction set and memory organization for the GPU.**

Keywords-Smith-Waterman; Graphics Processing Unit; Alignment; Residue

I. INTRODUCTION

The Smith-Waterman (SW) algorithm is a dynamic programming algorithm commonly used in computational biology for scoring local alignments between protein or DNA sequences. It guarantees the best possible local alignment [13, 12, 8]. An alignment score gives information regarding the similarity between one sequence and another. A protein sequence is essentially a string of characters. For example: "EITKFPKDQQNLIGQG" is a protein sequence containing 16 characters. Each character within a sequence is also known as a residue. The time complexity of SW is O(mn), where m and n are the lengths of the two sequences aligned [12, 4, 8]. Due to this computational requirement and the rapidly increasing size of sequence databases, faster heuristic methods such as FASTA and BLAST are often used as alternatives [7, 12]. FASTA and BLAST are at least 50 times faster than SW [3]. Heuristic methods may be faster than SW, however a price is paid with a loss of accuracy in finding the best possible local alignments. In this work we map SW onto the GPU architecture using Compute Unified Device Architecture (CUDA). CUDA is an extension of the C language developed by NVIDIA to allow programmer's access to their GPU's resources.

Our goal is to understand how a program's architecture, in the context of SW, fits onto a target processor's architecture; exploring ways to map the program architecture on the processor architecture. Our objective is to lay out a qualitative guidance as to how to judge the compatibility of a program's architecture and a processor's architecture based on the experiments carried out with this work. We also highlight the drawbacks of the GPU and its impact on mapping algorithms to the architecture. Our goal is to select the programming paradigm and its associated hardware platform that offers the simplest possible expression of an algorithm while fully utilizing the hardware resources. The GPU operates the same program code on several sets of data through the use of multiple threads. We test our work using a Tesla C870 GPU which uses NVIDIA's G80 architecture. Currently there is only one published attempt to map SW using CUDA by Manavski [7]. This method's performance relies on the use of CPU cores as computational muscle power in addition to the GPU. Our implementation is not tied to the use of CPU cores to supplement the GPU. Hence, our implementation can better portray the GPUs actual performance. We have achieved over 23 times speedup utilizing only the GPU. We further explore the memory organization and instruction set architecture of the GPU by analyzing another implementation on an Intel Quad Core processor that uses Intel's SSE2 architecture.

The rest of this paper is organized as follows: Section 2 describes the GPU architecture and the programming environment. Section 3 describes the Smith-Waterman algorithm. Related work is discussed in Section 4. Our mapping of the algorithm on the GPU architecture is detailed in Section 5. Section 6 contains our results. Section 7 presents the drawbacks of the GPU architecture, and Section 8 concludes the paper.

II. GPU/CUDA ARCHITECTURE

Until now, programming on a GPU required the user to reformulate algorithms and data structures using computer graphics primitives (e.g. textures, fragments) such as with OpenGL [5]. CUDA allows the programmer to program for Nvidia graphics cards with an extension of the C programming language. In CUDA, parallelized programs

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

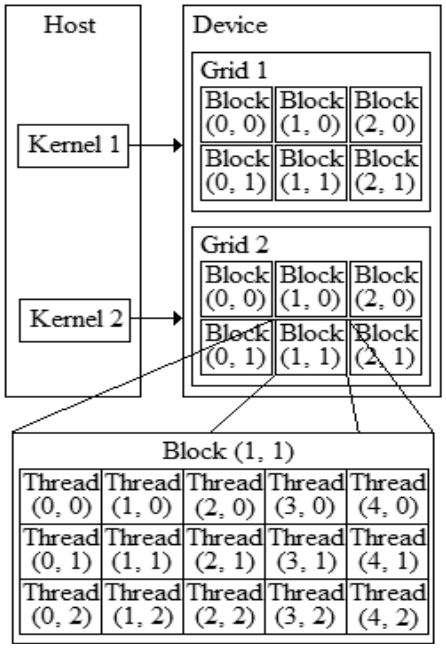

Figure 1. Structure of CUDA grid blocks

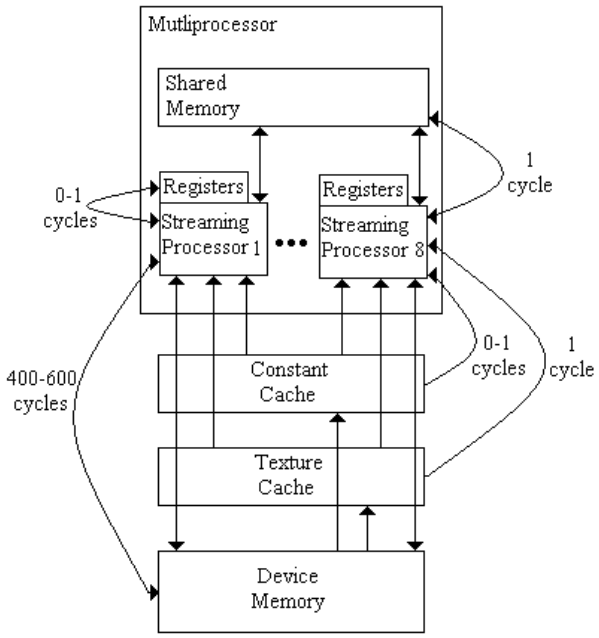

Figure 2. Structure of CUDA grid blocks

are launched from a kernel of code. The code on the kernel is run on several thousands of threads. Individual threads run all of the code on the kernel, but with different data. The kernel can be very small, or can contain an entire program. The threads running on the kernel are grouped into batches of 32, called warps. Batches of warps are placed within thread blocks. The device schedules blocks for execution on the multiprocessors in the order of their placement [10]. The placement is determined by the user, and is specified when launching the kernel from the host. Figure 1 illustrates the general layout of a grid of thread blocks, and Figure 2 illustrates the architectures' memory layout.

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

Each memory space is optimized for different memory usages [10]. The fastest memories are the shared memories and registers. These can be read from and written to by each thread. However, their size is severely limited. For each multiprocessor there is 16KB of shared memory, as well 8,192 32-bit registers. These registers and shared memory are on chip shared by all threads within a given multiprocessor. There can be up to 768 threads launched simultaneously per multiprocessor, which quickly use up these on chip resources [10]. The global memory, local memory, texture memory, and constant memory are all located on the GPUs main RAM. The texture memory and constant memory both have caches, and each cache is limited to 8KB per multiprocessor. The constant memory is optimized so that reading from the constant cache is as fast as reading from a register if all threads read the same address [10]. The texture cache is designed so that threads reading addresses with close proximity will achieve better transfer rates.

On our target GPU, the NVIDIA Tesla C870, there are a total of 16 multiprocessors. Each multiprocessor contains 8 streaming processors which operate at 1.35 GHz each, for a total of 128 streaming processors. The GPU has a thread manager which sends thread blocks to be executed in their respective order as determined by the user. In order to gain optimal performance when utilizing CUDA, the user must organize a program to maximize thread output, while managing the shared memory, registers, and global memory usage.

III. SMITH-WATERMAN ALGORITHM

The SW algorithm, besides being the most sensitive for searching protein databases for sequence similarities, is also the most time consuming [7, 12, 3]. 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. In Figure 3, we show an example of the progression of the algorithm when comparing two sequences. In Figure 3, the query sequence (sequence to be compared against the database) is "WHC" and the database sequence is "WCH". In this example cells

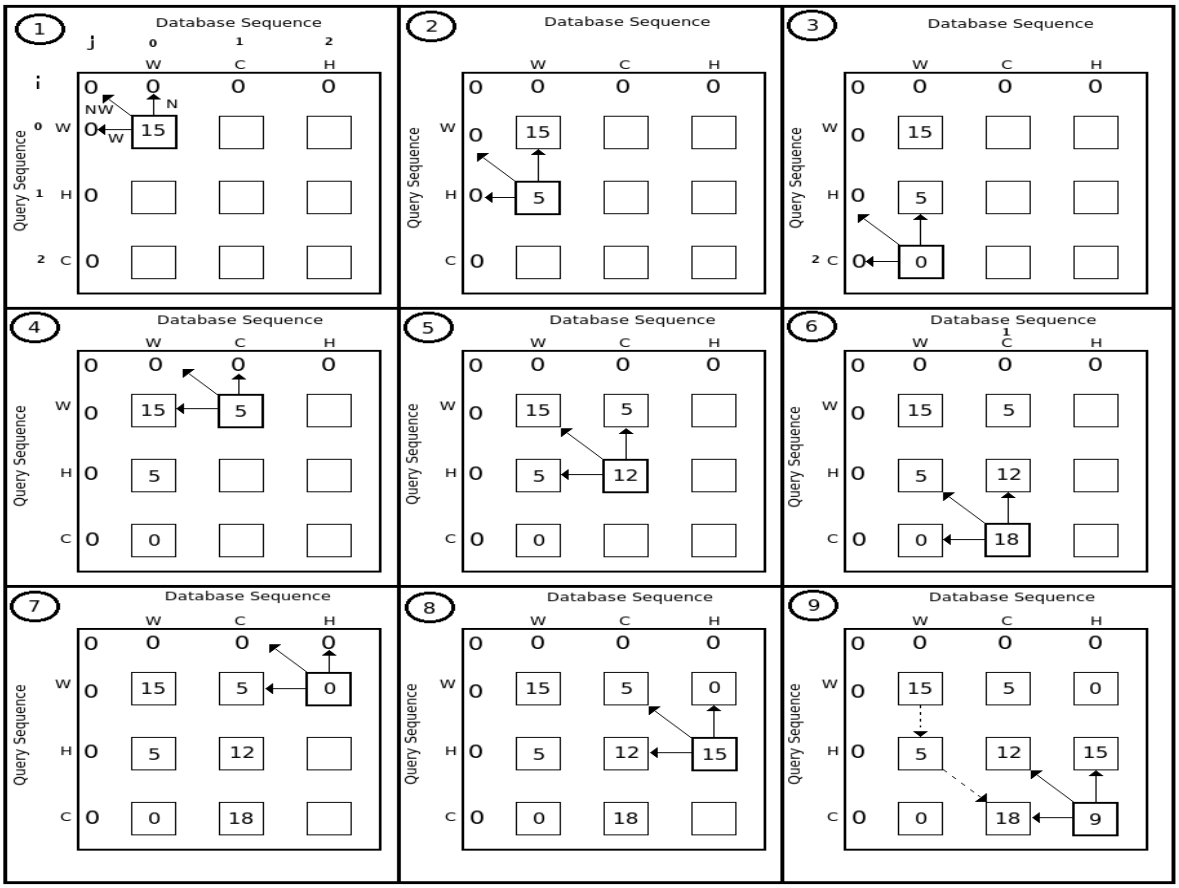

Figure 3. Smith-Waterman matrix filled in by iteration. A given matrix cell is dependent on north, north-west, and west cells. Each cell is filled with its respective score. This figure uses a BLOSUM50 substitution matrix.

within the "SW_matrix" are calculated by column, and dependencies are shown with arrows. The algorithm uses a lookup table to measure how similar two characters are. This lookup table is called a "Sub_Matrix", and is predetermined by the end user (Table 1).

$$H_{i,j} = \max \{ H_{i-1,j-1} + S_{i,j}, \quad \text{//north-west dependency}$$
$$H_{i-1,j} - G, \quad \text{//north dependency}$$
$$H_{i,j-1} - G, \quad \text{//west dependency}$$
$$0,$$
$$\} \qquad (1)$$

In (1), "i" is the current position in the "query_sequence", and "j" is the position in the "db_sequence". $H_{i,j}$ is the SW similarity score for "query_sequence[i]" and "db_sequence[j]". $S_{i,j}$ is a similarity score for a given residue combination as provided by a substitution matrix. A substitution matrix is a matrix which contains scores for every possible combination of residues. These scores are pre-determined by the life sciences community. If the maximum score for a given cell is based on the west or north cells, a gap is created thus causing a gap penalty (G) to occur. In the equation, a "zero" is added as the fourth parameter to prevent any cells in the matrix from going negative [1]. Figure 3 illustrates the nine iterations required to compute the similarity between the query sequence "WHC" and the database sequence "WCH". The outer walls of the matrix are padded with zeros so that boundary cells have starting comparison values. In iteration five of Figure 3 (i=1, j=1), the cell score is calculated as follows:

$S_{i,j}$ = Sub_Matrix[query_sequence[i]][db_sequence[j]];
$S_{i,j}$ = Sub_Matrix[H][C];
$S_{i,j}$ = -3;
G = 10;
$H_{i,j}$ = max {15 + (-3), 5 – 10, 5 – 10, 0 };
$H_{i,j}$ = 12 for SW_matrix score of current iteration

The inner loop of this algorithm (Figure 4) calculates the SW scores and cycles between the query sequence characters, and the outer loop iterates between the database sequence characters. Largest value in "SW_Matrix", 18 in this example, is the similarity score for the two sequences.

```
for (each database sequence)
  for (i=0;i<# of database_characters) {
    for (j=0;j<# of
query_sequence_characters){
          SW_matrix[i][j] = H;
    }
   Record max value in SW_matrix;
  }
}
```

Figure 4. Smith-Waterman Pseudo

TABLE I. PARTIAL BLOSUM50 SUBSTITUTION MATRIX

	C	H	W
C	13	-3	-5
H	-3	10	-3
W	-5	-3	15

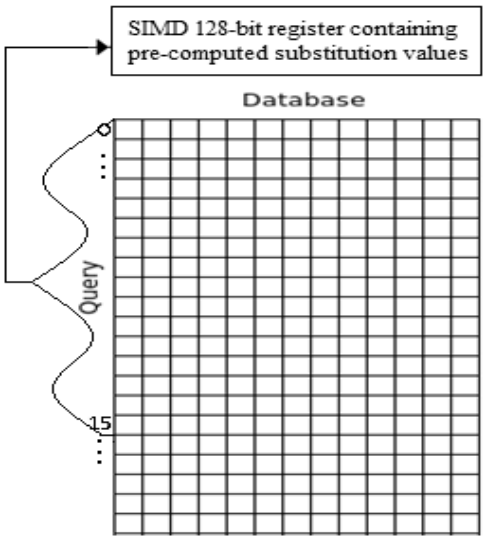

Figure 5. Accessing pre-computed substitution values from query sequence stored in SIMD registers in Farrar's implementation.

IV. RELATED WORK

A. *Farrar's SSE2 SIMD Approach*

Farrar presents an implementation in [2] which is written for Intel processors supporting SSE2 instructions. SSE2 instructions are a set of single instruction multiple data instruction sets (SIMD). These instructions allow for the usage of 128-bit wide SIMD registers, which Farrar utilized to speed up SW. In this implementation they create a query profile once for the database search. The query profile matrix contains all possible scores from the substitution matrix matched with the respective residues from the query matrix. By doing this they eliminate the need to perform any lookups of the substitution matrix during the internal cycle of the SW algorithm. The query profile is stored in 16 byte allocations, so that 16 elements from the profile matrix can be accessed with a single read from a SIMD register on the outer loop of the algorithm. Figure 5 illustrates how these values are accessed.

This gives them the ability process 16 cells at a time. If a cell's score happens to rise above 255, they recalculate with a higher precision by dividing the SIMD register into 8 16-bit elements. If the length of the remaining query becomes less than 16 elements, then they simply pad the remaining bits in the SIMD register with neutral values.

By accessing these 16 pre-computed elements with a single read from a register, they have greatly increased the execution speed of the algorithm. A buffer is used to store cells which future calculations will need, and is swapped with each iteration of the inner loop and replaced with new values.

B. *Manavski's CUDA Approach*

As of this writing there is only one published attempt to implement SW using CUDA. Manavski [7] implements SW on a GPU (NVIDIA GeForce 8800 GTX), and compares it against a serial version of SW and Farrar's implementation. Manavski uses a maximum query length of 567 characters in this implementation. It is at this length they achieve an execution time of 11.96(s) (30x speedup over serial version) using dual GPUs, and 23.32(s) using a single GPU. They report 20(s) of execution time for Farrar's SIMD implementation running on an Intel Quad Core processor.

For both single and dual GPU configurations, Manavski utilizes the help of an Intel Quad Core processor by distributing the workload among GPU(s) and the Quad Core processor. Manavski's design with a single GPU configuration performs worse than Farrar's implementation. They achieve only two times speedup with the dual GPU configuration against Farrar's implementation which only uses a single processor core. This indicates that their program architecture is not fitting on the GPU architecture well.

Manavski's method involves pre-computing a query profile matrix parallel to the query sequence for each possible residue similar to Farrar in order to reduce latency. Manavski sorts the database sequences according to length prior to running SW. The sorting is done so all threads within a thread block finish executing at approximately the same time. This implementation has the following issues:

- There is a major drawback in utilizing the texture memory of the GPU that leads to unnecessary caches misses. Just as Farrar, a query profile is pre-computed parallel to the query sequence for each possible residue. The query profile is stored it in the texture memory. With this method, the query profile quickly fills up the texture cache causing major memory delays due to cache misses. If the

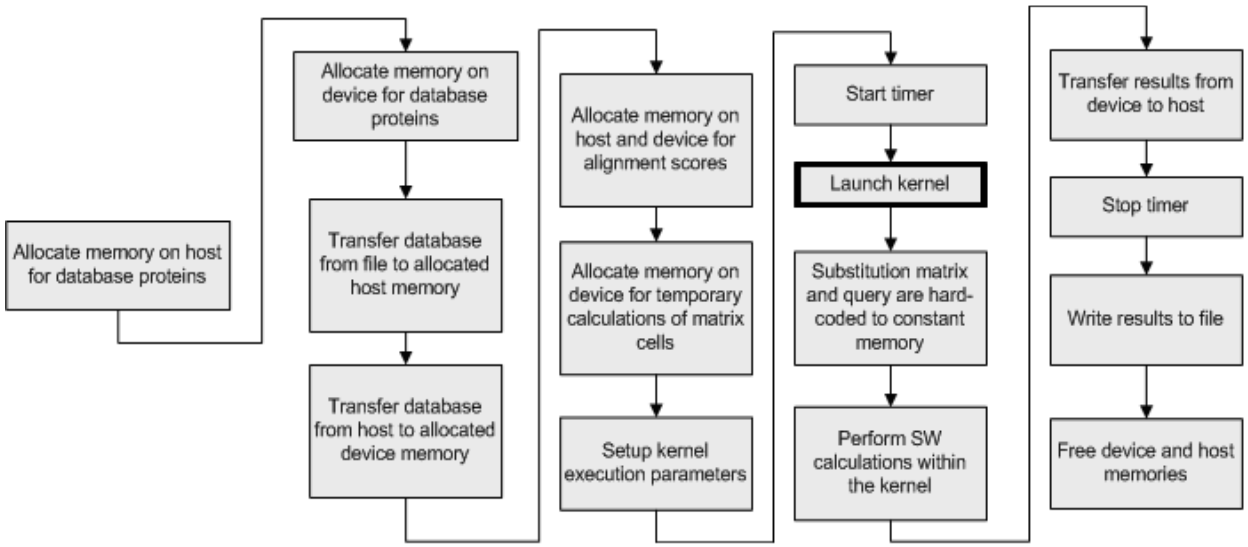

Figure 6. Top level program flow for GSW on host and GPU. The actual SW algorithm performs alignments on the GPU within the kernel (highlighted).

query length is larger than a length of 356, the query profile becomes larger than 8KB thus filling the texture cache. Therefore after a query length of 356, their implementation starts observing cache misses. Considering that query lengths can be much longer than 356, and each cache miss costs 400-600 cycles, with long query lengths GPU performance starts degrading significantly.

- The size of a grid of thread blocks is limited to 450. This makes it necessary for multiple launches of the kernel to process all database sequences, creating additional latency.
- Manavski also does not report any benchmarking data for query sequences above 567 residues running on their program. They say that it is possible to run their implementation with queries of lengths up to 2050, but since no data is provided for such lengths it is unclear how they would perform. This is not realistic since sequence lengths can vary greatly in size, and there is a good chance users would want to test queries above 567 residues. Moreover with query lengths being in the range of 1000's, their design will cause significant amount of cache misses on the texture memory.
- They report in their user's guide [6] that it is absolutely necessary to utilize additional CPU cores when the query sequences reach over 400 residues in length, or "serious problems could be encountered."

We conclude that Manavski's approach is highly CPU dependent and does not truly exploit the GPU architecture, because their program architecture does not overlap with the hardware architecture successfully. In the next section we introduce a new technique (GSW) that overcomes the drawbacks of Manavski's approach.

V. GSW GPU MAPPING

Unlike the Manavski method, we do not use the CPU for SW computations. We leave this strictly to the GPU to better understand its performance, and create a purely GPU dependent implementation. In Figure 6, we show the top level flow of our program. The SW algorithm is run on a kernel, as highlighted in Figure 6. A kernel represents the SW algorithm. Each streaming processor (128 of them) runs this kernel with its own data set. Figure 7 shows the flow chart of the kernel.

There are two cached memories on the GPU architecture, texture and constant. We have mapped our query sequence as well as the substitution matrix to the constant memory. We have done this because reading from the constant cache is as fast as reading from a register if all threads read the same address [10], which is the case when reading values from the query sequence. As mentioned, Manavski created a query profile matrix from the substitution matrix and the query sequence and stored this profile in the texture memory. Since this profile quickly creates a bottleneck due to cache misses, GSW is superior. We have used 64 threads per block, just as Manavski. In our implementation we calculated the SW score from the query sequence and database sequences by means of columns, four cells at a time due to the restrictions in the size of the shared memory. The progression of these calculations can be seen in Figure 8 for a database sequence of length 4 and a query of length 8. When cells are calculated, their updated values

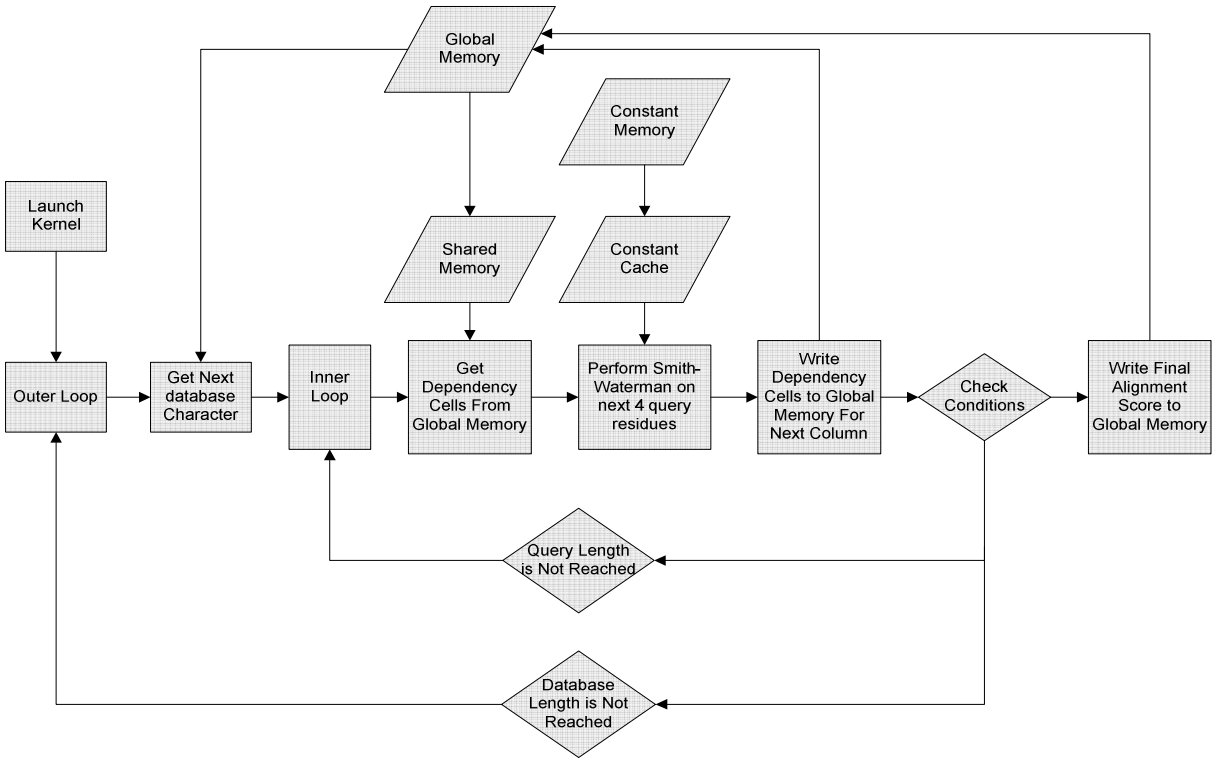

Figure 7. Flow chart of kernel execution. Outer loop switches between database residues, and inner loop switches query residues and performs Smith-Waterman calculations on 4 cells per iteration.

are placed in a temporary location in the global memory. This is represented by "cell calculations" in Figure 9. This cell calculations block is updated each time a new column is calculated, and is used for dependency purposes in calculating columns on each pass. These reads and writes to the global memory are extremely costly (400-600 cycles). However, there is simply not enough on chip memory to hold these values. For example: If there are 512 threads running on a multiprocessor, there would be 8,192 threads across all 16 multiprocessors. There would only be enough on chip memory to compute scores for queries which were less than or equal to a length of 4, which would completely fill out the on chip memory resources.

We keep track of the highest SW score for each thread in the kernel with a register, and it is updated on each pass of the inner loop. After the alignment is complete, the score is written to the global memory. The entire SW matrix calculations are not stored in memory, but rather just the temporary cell calculations column. This progression of calculations can also be seen in Figure 7, where we provide a flowchart of the kernel. Four dependency values are read at the beginning of the inner loop, and the new values for which the next column will be dependent are written at the end of the inner loop. The number of dependent cells needed for each alignment is simply equal to the length of the query sequence, since we are calculating cells by column. Currently our query lengths are limited to 1024 residues, but we are working on some indexing strategies which will allow us to increase the length of a query to 8,192 residues, which is also the size of the constant cache in Bytes. Even at queries of 1024 residues, this is nearly twice the number of residues presented by Manavski.

Since a substitution matrix is heavily utilized by SW, and a pre-computed query profile is not feasible on the GPU, we have come up with an efficient way of accessing it on the internal cycle of the algorithm. We placed the substitution matrix in the constant memory to exploit the constant cache, and created an efficient cost function to access it. Using the modulo operator is extremely inefficient on CUDA [10], so a more traditional hashing function could not be efficiently utilized for accessing the substitution values.

Common substitution matrices utilize 23 letters of the alphabet as residues and are symmetric, so the order of residues compared is irrelevant. We took this information and designed a simple yet extremely efficient cost function. Two changes need to be made to the substitution matrix for this function to work. The substitution matrix needs to be re-arranged in alphabetical order, and null characters must be inserted where there are unused characters of the alphabet. Generally there are only 3 unused characters which must be accounted for. The cost function can be seen in (2).

$$S_{ij} = (\text{ascii}(S1) - 65, \text{ascii}(S2) - 65) \quad (2)$$

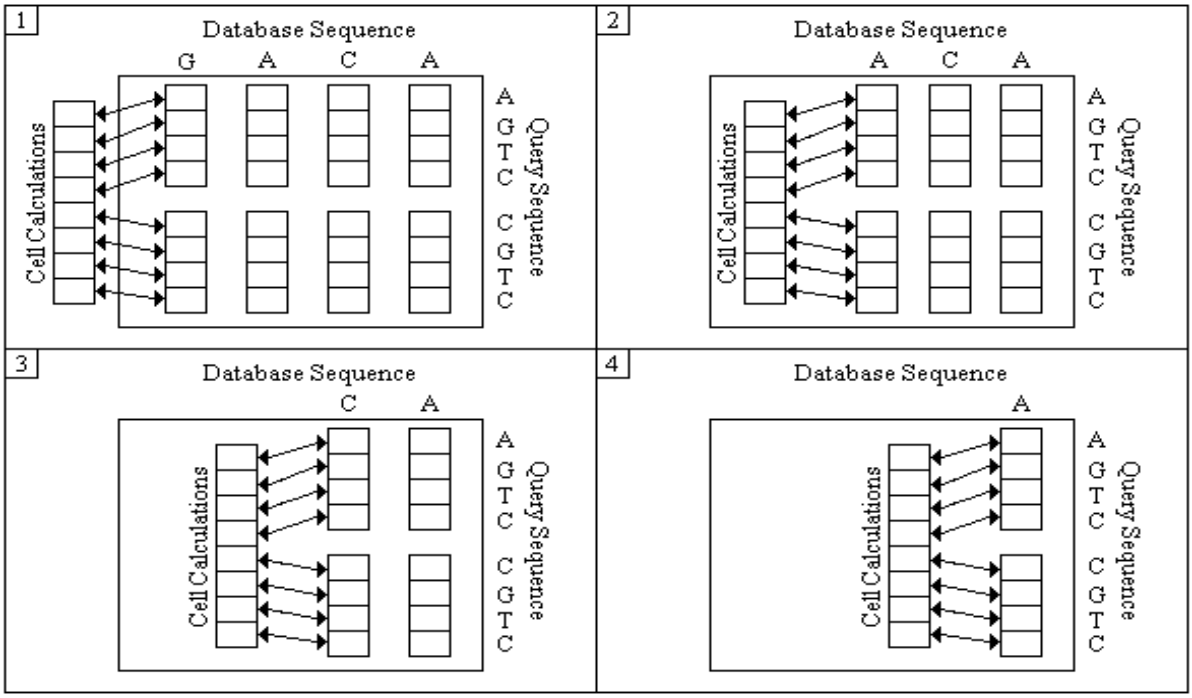

Figure 8. Example of calculation progression through a 4-residue database sequence aligned with an 8-residue query sequence. The cell calculation blocks are used to store temporary calculation values.

In (2), S1 is a residue from the query sequence and S2 is a residue from one of the Database sequences. By developing this function we are able to locate the exact position in the substitution matrix of the substitution value very efficiently. Table 2 contains a partial BLOSUM62 substitution matrix arranged in alphabetical order. The type of substitution matrix used is irrelevant to this procedure as long as 26 alphabetical characters are utilized and in alphabetical order. Since there are only 23 used characters in the BLOSUM62 matrix, null values need to be used as placeholders for this equation to work.

TABLE II. PARTIAL BLOSUM62 SUBSTITUTION MATRIX

	A (0)	B (1)	C (2)	D (3)
A (0)	4	-2	0	-2
B (1)	-2	4	-3	-3
C (2)	0	-3	9	-3
D (3)	-2	-3	-3	6

For example: if S1 is B, and S2 is D, then we will get the following from Equation (2), using Table. 1 for substitution values:

$$S_{ij} = (\text{ascii}(B) - 65, \text{ascii}(D) - 65)$$
$$S_{ij} = (66 - 65, 68 - 65)$$
$$S_{ij} = (1, 3) = -3$$
$$S_{ij} = -3$$

This equation allows us to take up a relatively small amount of space in the constant memory cache.

VI. RESULTS

We have tested GSW on a Dell T7400 running Red Hat Linux Enterprise v5.3. The computer has 2GB of RAM, and has dual 2.4 GHz Intel Quad Core Xeon processors. The computer also has the NVIDIA Tesla C870 GPU, which we are running our tests on. The amount of RAM and number of processors contained in our system is actually irrelevant, 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 information purposes. It does not have an effect on how the Tesla C870 performs its calculations.

Manavski's implementation requires mapping some of the workload onto the CPU for sequences above 400 residues in addition to the GPU. We feel that it serves no purpose to benchmark against their implementation for two reasons:

1) Speedup gained from the GPU side is not measurable: Implementation is not a true GPU mapping as it relies on CPU computations.

2) The program cannot provide alignments above 400 residues on the GPU: This restriction alone would cause omissions of several results from our tests.

We have run several tests with our implementation and compared it against, the serial version of SW called

TABLE III. COMPARISON OF OUR GSW IMPLEMENTATION AGAINST SSEARCH FOR PROTEIN LENGTH VERSUS PERFORMANCE IN TERMS OF EXECUTION TIME AND CLOCK CYCLE COUNT. THE DATABASE USED FOR ALIGNMENTS WAS SWISSPROT (AUG 2008), CONTAINING 392,768 SEQUENCES.

Protein ID	Protein Length	GSW Execution Time (s)	SSEARCH Execution Time (s)	Speedup	Clock Cycles SSEARCH (Billions)	Clock Cycles GSW (Billions)	Cycles Ratio
P36515	4	1.3	10	**8**	32.00	1.69	**18.96**
P81780	8	1.8	12	**6.8**	38.40	2.38	**16.11**
P83511	16	2.8	26	**9.2**	83.20	3.83	**21.71**
O19927	32	5.8	47	**8.1**	150.40	7.79	**19.30**
A4T9V0	64	11.2	99	**8.8**	316.80	15.17	**20.89**
Q2IJ63	128	22.0	212	**9.7**	678.40	29.64	**22.88**
P28484	256	43.8	428	**9.8**	1369.60	59.14	**23.16**
Q1JLB7	512	92.4	886	**9.6**	2835.20	124.78	**22.72**
A2Q8L1	768	144.6	1292	**8.9**	4134.40	195.15	**21.19**
P08715	1024	279.7	1807	**6.5**	5782.40	377.5	**15.32**

TABLE IV. COMPARISON OF OUR GSW IMPLEMENTATION AGAINST FARRAR FOR PROTEIN LENGTH VERSUS PERFORMANCE IN TERMS OF EXECUTION TIME AND CLOCK CYCLE COUNT. THE DATABASE USED FOR ALIGNMENTS WAS SWISSPROT (AUG 2008), CONTAINING 392,768 SEQUENCES. THE DOTTED LINE REPRESENTS THE POINT AT WHICH FARRAR'S IMPLEMENTATION OUTPERFORMS GSW.

Protein ID	Protein Length	GSW Execution Time (s)	Farrar Execution Time (s)	Speedup	Clock Cycles Farrar (Billions)	Clock Cycles GSW (Billions)	Cycles Ratio
P36515	4	1.3	2.78	**2.14**	6.67	1.69	**3.95**
P81780	8	1.8	2.81	**1.56**	6.74	2.38	**2.83**
P83511	16	2.8	2.96	**1.06**	7.10	3.83	**1.85**
O19927	32	5.8	3.35	**0.58**	8.04	7.79	**1.03**
A4T9V0	64	11.2	4.46	**0.40**	10.70	15.17	**0.71**
Q2IJ63	128	22.0	6.50	**0.30**	15.60	29.64	**0.53**
P28484	256	43.8	12.77	**0.29**	30.65	59.14	**0.52**
Q1JLB7	512	92.4	23.37	**0.25**	56.09	124.78	**0.45**
A2Q8L1	768	144.6	26.61	**0.18**	63.86	195.15	**0.33**
P08715	1024	279.7	37.47	**0.13**	89.93	377.5	**0.24**

SSEARCH. Our tests with SSEARCH were performed on a Gateway E-6300 containing a 3.2 GHz Pentium 4 processor, and 1 GB RAM. The Gateway system is running Windows Vista. We have considered the total clock cycles required to complete a database alignment as well as the total execution time of the program. We feel that this will give us a better comparison of results than simply reporting the raw execution times. SSEARCH uses a BLOSUM50 substitution matrix, a gap penalty of -10.

Our implementation was tested against SSEARCH using query sequence lengths ranging from 4 to 1024 residues. Table 3 provides all of our alignment results on the GPU vs performing alignments on SSEARCH. The average speedup in terms of execution time across all tests was 8.5 times. The greatest speedup achieved was 9.8 times while aligning a query with a length of 256 residues. To get the total number of clock cycles required for a given alignment, we took the clock speed of the processor and multiplied it by the execution time. For the CPU it was 3.2 GHz, and for the GPU it was 1.35 GHz. This allowed us to get more reasonable look at performance based on cycles to speed which is tied to the speed of the processor. On average our implementation has 20.22 times better performance in terms of clock cycles, and shows its peak performance while aligning a sequence of length 256 at 23.16 times.

We have also compared our implementation against Farrar's as shown in Table 4. We have tested Farrar's implementation on a system running Fedora 9 and with a 2.4 GHz Q6600 Quad core processor, however it should be noted that it is only utilizing a single core. Farrar's implementation was compiled with gcc. The gap penalty and gap extension penalties are set to -10 in Farrar's implementation as to create a linear gap penalty. After a query length of 32, Farrar's implementation outperforms GSW in terms of clock cycles.

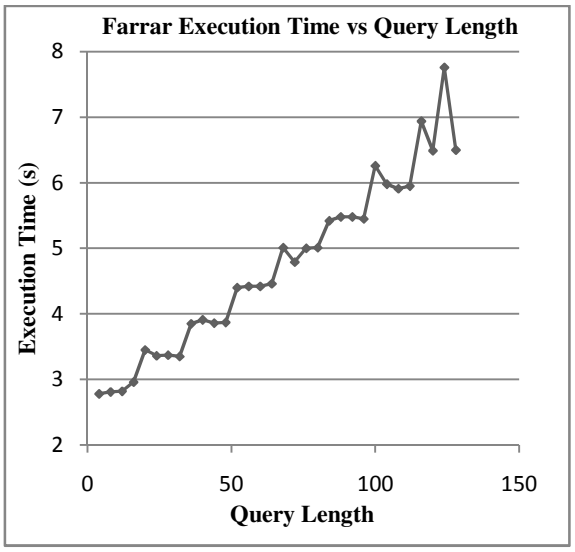

Figure 9. Farrar Execution Time, Database Used: Swissprot (Aug 2008) 392,768 sequences. Query lengths start at 4 and increase in increments of 4 to a length of 128.

VII. Drawbacks of GPU

Through this research we have found that the GPU architecture has issues that can have a negative impact on the execution of certain algorithms. The main drawback is the limited on chip memory. 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. This is extremely costly, 400-600 cycles, and creates a massive bottleneck. Since the calculations within a matrix for SW have dependencies on surrounding matrix cells, each thread must work on a separate alignment with a different database sequence. This is why so many reads are necessary from the global memory for the algorithm. The constant memory and texture memory can be helpful, however their 8KB caches are not sufficient for most applications and these memories are read only. Algorithms such as matrix multiplication and reduction algorithms are much more suited for the GPU architecture due to less severe data dependencies.

In Figure 9, we have further explored some tendencies of Farrar's implementation which makes use of Intel's SSE2 SIMD architecture. From the figure it can be clearly seen how reading the 16 pre-computed query profile values (totaling 128 bits) from the SSE2 register with a single read operation impacts the performance. The 128-bit SIMD register is completely filled with query profile values when the query length is a multiple of 16, and this is when Farrar gains optimal performance. This can be seen in Figure 9 (stepwise behavior), when the query is of lengths 16, 32, 48 and so forth. With the GPU we are only able to read 4 values at a time due to the restrictions on the shared memory, and these must be read from the global memory which cost 400-600 cycles.

Therefore, if reading blocks of 16 words at a time instead of 4 were possible from global memory into the shared memory then we would enhance the performance of the GPU significantly. Secondly, shared memory size is 16KB. We make use of it for "SW_Matrix" calculations (Figure 4). To be able to read 16 words at a time, we would also need 64KB of shared memory per multiprocessor as opposed to 16KB. This would allow us to access 16 characters from a query sequence at a time. These changes to memory organization would make GPU performance comparable to the SIMD based implementation.

VIII. Conclusion

Through this work we have found that SW can be effectively mapped to the GPU architecture using CUDA. We achieve a 23 times speedup over the serial method SSEARCH. Though we achieved a significant speedup, an SSE2 SIMD optimized version using a single core of an Intel processor achieves better performance due to its more accessible memory architecture for the algorithm. While the on chip memories are more than sufficient to perform the SW calculations in parallel, frequent accesses to global memory for reading data cause latency slowing down the program with its penalty of 400-600 cycles.

Advances in computing over the past several years have lead to an explosion in the amount of data generated in many scientific fields, including life sciences. The complexity, variety of needs, and storage demands associated with the applications of biological systems pose several challenges particularly from the points of scalability and resource utilization. This immense increase in data is forcing scientists to explore alternative high-performance computing platforms. From that perspective, with its computation power, GPU architecture poses as an ideal candidate when equipped with domain specific instruction set support and faster access from shared memory to the global memory. Adding additional instructions for the architecture such as multiply and accumulate (MAC) and fused multiply add (FMA) would make it more suitable for specific applications. The source code for our implementation for GSW can be found at http://ece.arizona.edu/~rcl/gsw.html.

Acknowledgment

We would like to thank Nirav Merchant and Dr. Susan J. Miller from the Arizona Research Labs for their input and guidance on the Smith-Waterman algorithm and its applications.

REFERENCES

[1] Eddy, S.R.: Where did the BLOSUM62 alignment score matrix come from? Nature Biotechnology. 22:1035-1036 (2004)

[2] Farrar, M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23, pp 156, 161 (2007)

[3] Walker, J.M.: The Proteomics Protocols Handbook, Humana Press, pp 504 (2005)

[4] Liao, Y.H., Yin, L.M., Cheng, Y.: A parallel Implementation of the Smith-Waterman Algorithm for Sequence Searching. In: Proceedings of the 26th Annual International Conference of the IEEE EMBS (San Francisco, California, September 01-05, 2004)

[5] Liu, W., Schmidt, B., Voss, G., Schröder, A., Müller-Wittig, W.: "Bio-Sequence Database Scanning on a GPU," In: Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (HICOMB Workshop). (2006)

[6] Manavski, S.S.: Smith-Waterman User Guide. http://bioinformatics.cribi.unipd.it/cuda/docs/SWCudaUserGuide.pdf (2008)

[7] 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)

[8] Mount, D.W.: Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, pp 64-65, 71-87. (2004)

[9] NCBI: National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt. May 2008. (2008)

[10] NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide. http://www.nvidia.com/object/cuda_develop.html, (2008)

[11] NVIDIA Corporation: NVIDIA Tesla C870 overview. http://www.nvidia.com/object/tesla_c870.html, May 2008. (2008)

[12] 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)

[13] Smith, T., Waterman, M.: Identification of common molecular subsequences. Journal of Molecular Biology. V147. 195-197. (1981)

[14] UVA: FASTA program webpage. http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml (2008)