A Parameterisable and Scalable Smith-Waterman Algorithm Implementation on CUDA-compatible GPUs

Cheng Ling[1], Khaled Benkrid[1] and Tsuyoshi Hamada[2]

[1]*Institute for Integrated Micro and Nano Systems, Joint Research Institute for Integrated Systems, The University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, UK*
[2]*Faculty of Engineering, Department of Computer and Information Sciences, Nagasaki University, Bunkyo-machi, Nagasaki 852-8521, Japan*
[1]*C.Ling@ed.ac.uk,* [1]*K.Benkrid@ed.ac.uk,* [2]*hamada@cis.nagasaki-u.ac.jp*

***Abstract*—This paper describes a multi-threaded parallel design and implementation of the Smith-Waterman (SM) algorithm on compute unified device architecture (CUDA)-compatible graphic processing units (GPUs). A novel technique has been put forward to solve the restriction on the length of the query sequence in previous GPU implementations of the Smith-Waterman algorithm. The main reasons behind this limitation in previous GPU implementations were the finite size of local memory and number of threads per block. Our solution to this problem uses a divide and conquer approach to compute the alignment matrix involved in each pairwise sequence alignment, as it divides the entire matrix computation into multiple sub-matrices and allocates the available amount of threads and memory resources to each sub-matrix iteratively. Intermediate data is stored in shared and global memory on the fly depending on the length of sequences in hand. The proposed technique resulted in up to 4.2 GCUPS (Giga Cell Updates per Second) performance when tested against the SWISS-PROT protein database, which is up to 15 times faster than a equivalent optimised CPU-only implementation running on a Pentium4 3.4GHz desktop computer. Moreover, our implementation can cope with any query or subject sequence size, unlike previously reported GPU implementations of the Smith-Waterman algorithm which makes it fully deployable in real world bioinformatics applications.**

1. INTRODUCTION

Biological sequence alignment is a widely used operation in the field of bioinformatics and computational biology. It aims to find out whether two or more biological sequences are related or not. However, biological sequence alignment is also a computationally expensive application as its computing and memory requirements grow quadratically with the size of the databases. Given that the latter is growing exponentially year after year, the need for hardware acceleration is getting stronger [1].

Graphics Processor Units (GPUs) have been proposed recently as a high performance and relatively low cost acceleration platform for biological sequence alignment [2]. Among the early attempts we can cite Liu's OpenGL-based implementation of the Smith-Waterman algorithm, reported in [3]. More recently, Manavski et al. [4] and Munekawa et al. [5] reported two different GPU implementations of the Smith-Waterman algorithm using NVIDIA GPUs [6]. The former used a single thread to compute a complete pairwise alignment matrix, column by column serially, and harnessed many threads to compute the alignment matrices of different pairs in parallel. The latter implementation harnessed one batch threads (block) to compute a single alignment matrix in parallel, exploiting the fact that the computation of the matrix cells on each anti diagonal are independent of each other, and hence can be done in parallel. Both implementations used the compute unified device architecture (CUDA) API to program GPUs and targeted GPUs from NVIDIA [8]. These API functions have contributed greatly to the use of GPUs in general purpose computing, opening the way for a new field of computational study coined general-purpose computation GPU (or GPGPU) which aims at harnessing GPUs for a wide range of applications including scientific computing [7], computational geometry [8], image processing [9] and bioinformatics [2].

Compared with CPU-based implementations of the Smith-Waterman algorithm e.g. from Farrar [10], Manavski et al. [4] and Munekawa et al. [5] demonstrated good acceleration performance which runs from 2 to 30 times faster than any previous implementations on commodity hardware. However, both implementations have a serious limitation on the length of query sequences that their GPU implementations can cope with. Indeed, Munekawa et al. [5] is clearly stated that that query sequences should be shorter than 2048, because of the limitation in the maximum number of threads that could be defined in each block. Moreover, Manavski et al. [4] reports a similar limitation because of the limited size of the local memory. Such limitation renders these implementations useless in many real world applications where query sequences are far longer than 2500 approximately. This paper presents a technique which overcomes the above limitation of previous implementations of the Smith-Waterman algorithm. The main idea behind this is to separate the computation of the alignment matrix into multiple parts if the number of threads and size of local memory are not sufficient, and allocate the available resources to each sub-matrix in turn.

The remainder of the paper is organized as follows. First, relevant background on the Smith-Waterman algorithm is presented. Then, previous work in the area of GPU-based acceleration of biological sequence alignment is presented. After that, our novel GPU-based implementation technique of the Smith-Waterman algorithm is presented. A comparative evaluation of our implementation then follows before conclusions and ideas for future work are laid out.

2. THE SMITH-WATERMAN ALGORITHM

The Smith-Waterman algorithm [11] is dynamic programming algorithm which finds the best local alignment between two sequences. The optimal local alignment obtained by the algorithm is achieved in two stages. Firstly, an alignment matrix is calculated based on the correlation between the two sequence characters (e.g. protein amino acids, DNA base pairs). The optimal local alignment is found by finding the maximum element in the alignment matrix, which attaches a score to the degree of similarity between the two sequences, and tracing back the alignment matrix until a zero element is found.

More specifically, let D denotes a database sequence of length *m*:

D: $d0d1d2d3.....dm-1$

Let Q denotes a query sequence of length *n*:

Q: $q0q1q2q3.....qn-1$

Let $W(ai,bj)$ denotes the substitution scoring matrix which gives a score describing the likelihood of substitution between characters a_i and bj.

Let $Ginit$ and $Gext$ denote penalties for opening a new gap and continuing an existing gap respectively.

With the above, the alignment matrix computation of the Smith-Waterman algorithm is described by the following equations:

1. $Ei,j = \max\{Hi,j-1-Ginit, Ei,j-1-Gext\}$
2. $Fi,j = \max\{Hi-1,j-Ginit, Fi-1,j-Gext\}$
3. $Hi,j = \max\{0, Ei,j, Fi,j, Hi-1,j-1+W(ai,bj)\}$

The values of Hi,j, Ei,j and Fi,j are defined as 0 if $i<1$ or $j<1$ .The gap penalty is called linear if $Ginit = Gext$, otherwise, it is called affine. In our subsequent GPU implementation, we use a linear gap model, which means that the above equations can be summarised as:

4. $Hi,j = \max\{0, Hi-1,j-G, Hi,j-1-G, Hi-1,j-1+W(ai,bj)\}$

From this equation, we observe that the value of Hi,j depends on the values of its upper neighbour $Hi,j-1$, left neighbour $Hi-1,j$ and left-upper neighbour $Hi-1,j-1$, as shown in figure 1.

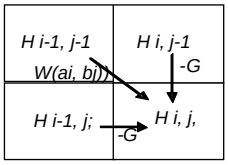

Fig1. Data dependency of the Smith-Waterman dynamic programming algorithm

The above operations are massively parallelisable since the anti diagonal elements of the alignment matrix are independent of each other, and hence can be computed in parallel. In addition, the computation of different alignment matrices between a query sequence and several subject sequences can be done in parallel too. Since GPUs have the ability to allocate thousands of parallel threads to a particular task, it is a very appealing acceleration platform for the Smith-Waterman algorithm. Note finally that the trace back procedure will be done only for one or few subject sequences, the one(s) with the highest score, out of thousands or millions of subject sequences, and hence it is better achieved on the host CPU. The GPU parallelisation task is hence focused on the alignment matrices' calculation.

3. THE PROGRAMMING MODEL OF CUDA-COMPATIBLE GPUS

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA Corporation [6], which makes the computing engines of graphics processor units accessible to general purpose software developers through a standard programming language e.g. C, with an API to exploit the architecture parallelism. Like many-core CPUs, CUDA uses threads for parallel execution. However, whereas multi-core CPUs have only few threads running in parallel at any particular time, GPUs allow for thousands of parallel threads to run at the same time. The GPU used in our implementation is NVIDIA's Geforce 8800GTX (see Figure 2) which has 16 Stream Multiprocessors (SMs), with each SM having eight Stream Processors (SPs) used as Arithmetic Logic Units (ALUs) with 8KB constant cache, 8KB texture cache and 16KB shared memory (see Figure 2). The shared memory can be read and written by any thread in a block assigned to a SM. In addition, each SP has its own registers (1024) and operates the same kernel code as the other SPs, but with different data sets. Access speed to shared memory is as fast as accessing SP registers as long as there are no bank conflicts [6].

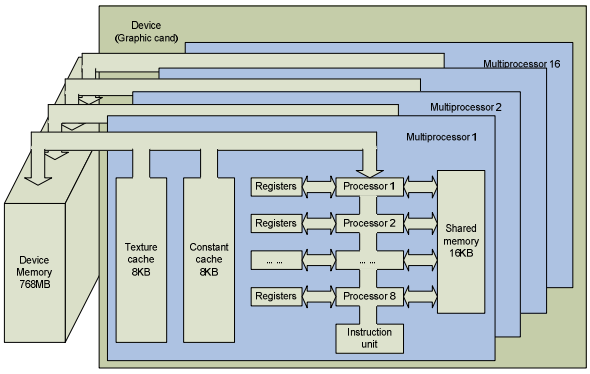

Fig2. Block architecture of NVIDIA's Geforce 8800 GTX

In addition to the above, a device memory offers global access to a larger (768 MB) but slower storage. Any thread in any SP can read from or write to any location in the global memory. Since computational results can be transferred back to CPU memory through it, global memory can be thought as a bridge which achieves communication between GPU and CPU.

Local memory is allocated automatically if the size of variable required is bigger than the register size, and there is no keyword specified for it. It is not cached and cannot be accessed in a coalesced manner like global memory. Texture memory within each SM can be filled with data from the global memory. It acts like a cache, and so does constant memory, which means that they can speed up the fetch time of data. However, threads running in the SMs are restricted to read only access to these memories. The host CPU, on the other hand, does have write access to these memories.

4. OUR SMITH-WATERMAN GPU IMPLEMENTATION

Part of our proposed technique for the Smith-Waterman GPU implementation draws from the experience of Liu's parallelization strategy reported in [3] and the memory distribution scheme of Munekawa et al. reported in [5]. Before reporting the details of our novel technique, we first describe the two main parallelization strategies adopted for the GPU acceleration of the Smith-Waterman algorithm. Afterwards, we will illustrate our improved strategy which solves the problem of query size limitation reported in previous implementations.

A. Parallelization Strategies

Eqs.4 indicates that the computation of matrix cell $H_{i,j}$ just depends on the values of its upper neighbour $H_{i,j-1}$, left neighbour $H_{i-1,j}$ and its left-upper neighbour $H_{i-1,j-1}$, which means that the calculation of cells within each anti diagonal of alignment the matrix can be done in parallel.

Obviously, for the computation of cells on the k-th anti diagonal, we need to record the cells on (k-1)-th anti diagonal and the cells on the (k-2)-th anti diagonal. For a query sequence of length *n* and a database subject sequence of length *m*, there are $m+n-1$ anti diagonals which have to be computed serially. Instead of storing all matrix cells, we just need to allocate memory for the storage of two anti diagonals. As illustrated in Figure 3, the computation of cells in the i-th row (i>1) depends on both the value stored in shared[i-1] and shared[i]. Moreover, shared[i] will be updated when the new H value is computed for the computation of the cells in the i-th row. Therefore, we use a register for each thread to store the cells on the (k-2)-th anti diagonal and shared memory space for the cells on the (k-1)-th anti diagonal. After computing all cells on the k-th anti diagonal, we use the cells on the (k-1)-th anti diagonal to update the content of registers and the cells on current k-th anti diagonal to update the content of shared memory for the computation of all cells on (k+1)-th anti diagonal. Another register defined in kernel for each thread is used to store and update the highest score of each row. Overall, one block of threads is responsible for computing one matrix, and each thread contained in it takes charge of the computation of one row. However, due to the limitation of the maximum number of threads which is 512 for each block, a problem occurs when the query sequence size is longer than the maximum number of threads possible. Considering this limitation, Munekawa et al. [5] utilize built-in variable char4 [6] for each thread to expand the maximum query length possible to 2048 (=512*4). Nonetheless, the problem remains for longer query sequences. To circumvent the maximum number of threads limitation, Manavski et al. [4] proposed a different parallelization strategy. In it, one single thread is allocated to the computation of the entire alignment matrix of a pair of sequences. The alignment matrix is calculated serially column by column (see in figure 4). Since the computation of each column just depends on the previous one, a small amount of memory size is needed, dependent on the length of query sequence.

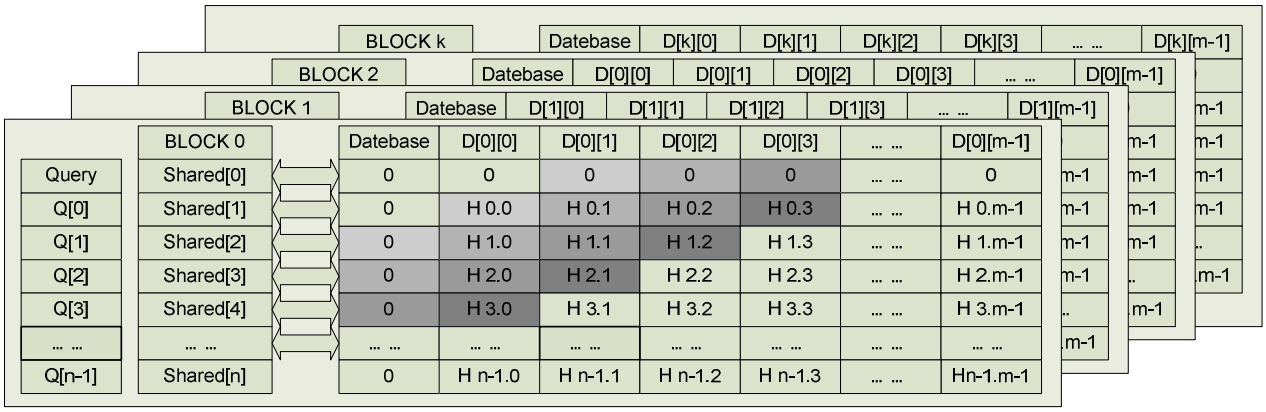

Fig3. Parallel implementation of the alignment matrix computation for k pairs of sequences - the equally shaded parts stand for anti diagonal cells that can be computed in parallel

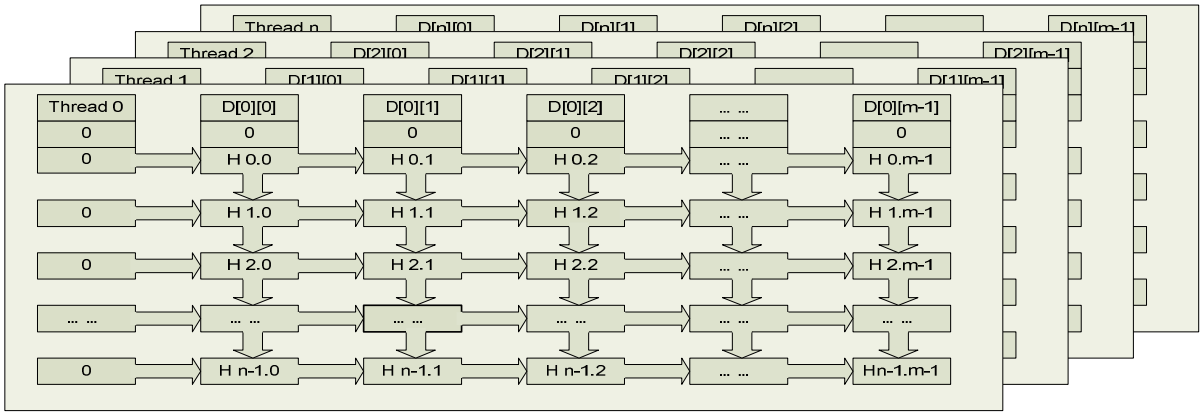

Fig4. Parallel implementation of the alignment matrix computation – one single thread computes a complete alignment matrix of two sequences. Cells in each matrix are computed column by column using local memory as a buffer for temporary data

Though the computation of cells of each column is serial for each thread, massive parallel computation is achieved by using multiple parallel threads to calculate alignment matrices between a query sequence and several subject sequences. In this implementation, however, the amount of parallelism is restricted by the size of the local memory shared by all parallel threads which is used to store and load temporary data necessary for the calculation of alignment matrix elements. For different number of parallel threads, the available local memory allocated for each thread is not fixed, which will be decreased if more threads are running at the same time. As a result, this method also suffers from a limitation in the maximum possible size of query length to be processed. In next section, we will describe our proposed method for solving this problem.

B. Our Thread Allocation and Reuse Strategy

The parallelization strategy adopted in our design is similar to the approach adopted by Munekawa et al. [5] as we allocate several threads to compute a single alignment matrix. However, in our implementation we separate a single alignment matrix computation into multiple sub-matrices with a certain number of threads, commensurate with the maximum number of threads and the possible maximum amount of memory available. Once the batch of threads allocated completes a sub-matrix calculation, the final thread in the batch records the data in the row which it takes charge and stores it into shared memory or global memory depending on the size of database subject sequence, ready for the calculation of the next sub-matrix, and the first thread in the batch loads this data as initial data for the subsequent sub-matrix calculation.

This operation continues in turn until the end of the entire alignment matrix calculation. This process is illustrated in Figure 5 as below where the final thread in each batch (thread *n*) stores the cells of its row. Afterward, the first thread in the batch loads these values as initial data for the computation of the first row in the next alignment sub-matrix. It is worth mentioning here that all alignment matrix calculations are done purely on GPU. The host processor only allocates memory on the GPU device and predefines the relevant database sequences offset which guarantees that each block operates on right section of the database sequence before launching the GPU kernel.

Since each SM can have 768 parallel threads running at the same time, we split this amount into batches of threads or blocks, where each block computes one alignment matrix. For example, we can split the overall

number of threads into 8 blocks of 96 threads, with 10 registers allocated to each thread and each block could use almost 2 KB of shared memory. Global memory will be used if this amount of allocated shared memory space is not enough for any database subject sequence. Note here that if the length of the database subject sequence is smaller than the number of thread in the block, additional waiting time should be added for the threads in the batch to finish their computations. This is easy to imagine e.g. if thread 0 has already completed its row calculation, but thread *n* has not completed yet or has not even started its row, then thread 0 of the next block would have to wait for thread *n* of the previous block to complete its task before obtaining its initial data for the next batch of processing i.e. row *n+1*. The waiting time is proportional to the number of threads minus the length of database subject sequence if the length of database subject sequence is smaller than the maximum thread number, otherwise, it is 0.

C. Load partitioning and speed-up strategy

In our implementation, we use constant cache to store the commonly used constant parameters in order to decrease access time, including the substitution matrix and the query sequence. In addition, we use global memory to store the database sequence as the size of the latter can be in the hundreds of megabytes. Moreover, we use texture cache to shade database sequences. The bottleneck of speedup in our implementation is the store operation of temporary data by the last thread and the load operation by the first thread in each batch, because the latency between SP registers and global memory is much longer than the one between registers and shared memory. No matter how fast other threads execute the kernel code, they have to wait for a point where all threads synchronize. Obviously, this only occurs when the length of the database subject sequence is longer than the allocated space in shared memory. Therefore, our acceleration strategy mainly focuses on the efficient allocation of resources to each block to make the maximum use the available parallelism. This can be achieved through setting the number of threads in each block.

Since each SM has 8192 registers and can keep at most 768 threads running at the same time, for a query sequence of length 512, if we use 1 block of 512 threads, 16 registers can be used for each thread. In this case, only one sequence alignment can be computed in each SM. If we use 8 blocks of 64 threads, also 16 registers can be allocated to each thread, but the number of sequence alignments can be processed at the same time becomes to 8. Rather than adopting the simple method used by Munekawa et al. [5] which utilizes the full memory resource for each block, we flexibly allocate resources through setting the number of threads in each block, with no limitation on the overall length of the query sequence. Table 1 presents execution times of the Smith-Waterman algorithm on GPU using our technique, with different numbers of threads per block. For a query sequence of length 1023, 64 threads per block lead to the best performance.

Table1. Performance comparison among thread number of 64, 128 and 256. All query sequences run against the SWISS-PROT database [12]

Query length	Thread 64	Thread 128	Thread 256
	Time(sec)	Time(sec)	Time(sec)
63	2.13	3.1	6.18
127	6.11	4.16	7.15
191	9.28	11.94	8.3
255	12.45	12.93	9.63
511	25.12	26.35	29.17
1023	50.4	53.1	57.8

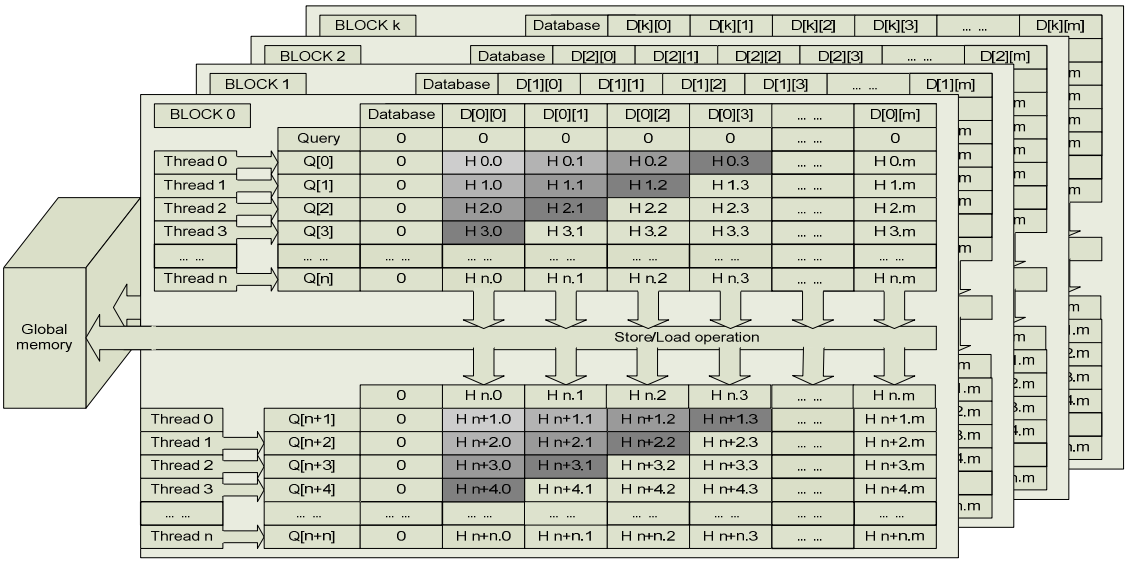

Fig5. Our thread reuse strategy, store and load operations are performed by the final thread and the first thread in each thread batch (block).

In our implementation, we use vector type *char2* as illustrated in Figure 6 to decrease the data fetch times compared to using *char*. This was empirically found to be more efficient than using vector *char4*.

Thread 0	Q[0]	D[0], D[1]	D[2], D[3]	D[4], D[5]	… …	D[m-1], D[m]
Thread 1	Q[1]	D[0], D[1]	D[2], D[3]	D[4], D[5]	… …	D[m-1], D[m]
Thread 2	Q[2]	D[0], D[1]	D[2], D[3]	D[4], D[5]	… …	D[m-1], D[m]
… …	… …	… …	… …	… …	… …	D[m-1], D[m]
Thread n	Q[n]	D[0], D[1]	D[2], D[3]	D[4], D[5]	… …	D[m-1], D[m]

Fig6. Alignment matrix calculation with vector variable char2

D. Pseudo Code

Figure 7 shows the pseudo code for the proposed kernel. According to the length of query sequence and the amount of threads, firstly, we compute the reuse rate of threads in line 3, and then compute the additional waiting time for the purpose of insuring the proper initial data for the computation of the next iteration in line 5. Each thread reads the corresponding cells on the query sequence in line 7. Note that there is a stringent requirement for the computation order among threads, which is controlled by the variable key in line 12. When *key* < 0, this indicates that the thread still needs to wait (until *key* = 0). After that, it starts to fetch cells in database sequence to perform the necessary computations. Since we define vector variable *blockdb* in line 17, the maximum fetch time can be decreased from *dbfetch* to *dbfetch*/2. When *key* > *dbfetch*/2, this indicates that the thread has already fetched the final cells and there is no need to do any extra operations it line 13. In the final iteration, some threads may not have any pending tasks, in which case they just wait for the synchronization operation in line 30. Line 18-20 show the computation of the alignment values and the update of the maximum score value. After that, new anti diagonals are updated by changing the value of variable *dia* in line 21 and the corresponding positions in shared memory in line 22. A similar procedure is used in lines 23-27 but with different database sequence cells. Note here that we adequately use the H value stored in registers for computation instead of using shared memory, which avoids the risk of data update conflict among threads and improve operation speed. Apart from the computation duty, the final thread (*tid* = *threadnum*-1) stores H for the purpose of the next iteration in line 28. Note that if *fetch*>0, *thread 0* also needs to load the initial data in line 16. Load and store operations are added at the beginning and the end of the inner loop. Finally, every thread copies the highest score of its rows (*threadnum***fetch* + *tid*+1) to global memory in line 30. The expression in *s* guarantees that there is no conflict in device memory writes.

```
Input:  Query sequence __constant__ query[Len_query];
        Database sequence dbTex[offset[bid]];
        Substitution matrix scoring[400];
Output: Highest scores s[bid*(Len_query/threadnum+1)];
1.  __shared__ int anti_x[threadnum+1]; //(k-1)-th anti diagonal
2.  __shared__ int anti_y[threadnum+1]; // k-th anti diagonal

3.  quefetch=Len_query/threadnum+1; //thread reuse time
4.  dbfetch=offset[bid]; //different database sequence
5.  wait=|dbfetch-threadnum|;
6.  for fetch=0 to quefetch-1 do //for all query cells
7.     subque=query[tid+threadnum*fetch];
8.     dia=0; //(k-2)-th antidiagonal
9.     anti_x[tid]=-2;  //tid: thread ID
10.    anti_y[tid]=-2;
11.    for times=0 to dbfetch/2+threadnum+wait do
12.       key=times-tid;
13.       if key<0 or key>dbfetch/2 then wait or finished;
14.       else if tid>Len_query-(fetch*threadnum)-1 then end;
15.       else
16.          if fetch>0 and tid=0 then load initial data; end if;
17.          blockdb=text1Dfetch[key]; // fetch cells in database

18.          compute w(subque, blockdb.x);
19.          H=max(0, dia+w, anti_x[tid], H-2);
20.          score=max(score, H);
21.          dia=anti_x[tid]+2;
22.          anti_x[tid+1]=H-2;

23.          compute w(subque, blockdb.y);
24.          H=max(0, dia+w, anti_y[tid], H-2);
25.          score=max(score, H);
26.          dia=anti_y[tid]+2;
27.          anti_y[tid+1]=H-2;

28.          if fetch>0 and tid=threadnum-1 then store H; end if;
29.       end if;
30.       __syncthreads(); //synchronization
31.    end for;
32.    s[bid*threadnum*quefetch+fetch*threadnum+tid]=score;
33. end for;
```

Fig7. Pseudo code. Each block executes the code with the same query sequence and different database sequences; the output is the maximum score for each row in each matrix. tid and bid represent the ID of each individual thread and block respectively. Gap penalty is linear and equal to 2. The constant substitution matrix is stored in constant memory.

5. Results and Discussion

In this section, we present experimental results of our Smith-Waterman GPU implementation compared to the state-of-the-art. In our implementations, we used a Mac Pro desktop computer running Ubuntu 8.10 32-bit Linux operation system, with an NVIDIA GeForce 8800 GTX GPU with 768MB device memory, with 576MHz core clock frequency, and 900MHz memory clock frequency. We used NVIDIA SDK 9.5 and CUDA 1.1 API for our code development.

The experiments reported used query sequences of lengths ranging from 63 to 511 amino acids. All query sequences run against the Swiss-Prot protein sequence database [12], which approximately 180MB in size, and contains 399,749 sequence entries with a total of 144041553 amino acids. The execution times reported are just for the execution time on GPU.

The version of the Swiss-Prot database used in Liu et al's OpenGL method [3] is release 46.3, March 2005. It

contains 176,469 sequence entries, with an average length of 361 amino acids. The GPU type used in their implementations was an NVIDIA Geforce 7800GTX GPU. For the sake of simplicity, they used a simple substitution matrix which uses a score of 2 if the characters from database sequence and query sequence are identical and -1 otherwise. For our implementation, the performance was tested using the more biologically accurate BLOSUM 50 substitution matrix. Moreover, we tested the performance with a simple version substitution matrix for the purpose of fair comparison with Liu's method [3], the results of which are shown in Table 2. The evaluation of our implementation on the Geforce 8800GTX GPU shows a speedup factor from 4x to 20x. However, the two implementations used GPUs from different generations. In order to allow for a fairer comparison, we compared our implementation with more recent GPU implementations from Munekawa et al. [5] in table 3 and Manavski et al. [4] in table 4.

Table2. Performance comparison between Liu's openGL method [3] and our proposed method, both using a simple substitution matrix

Query length	Execution time T (sec)		Throughput P (MCUPS)	
	Proposed	Liu's	Proposed	Liu's
63	2.13	19.5	4059	196
127	4.16	25	4189	308
255	9.63	36.3	3634	427
511	25.12	59.2	2792	524
1023	50.4	105.1	2786	591
2047	101.12	197.9	2778	628
4095	202	383.1	2782	649

Table 3 presents comparative implementation results between our implementation and Munekawa et al.'s which targeted NVIDIA's Geforce 8800GTX GPU [5]. Here, we can see that for shorter query sequences, our implementation performs better, but as query sequence length increases, the performance of our implementation decreases. This is because of the overheads associated with storing and loading intermediate data between computation batches when the query sequence length is greater than the number of threads, as explained in the previous section. Nonetheless, we notice that Munekawa et al.'s implementation cannot cope with query sequences longer than 2048, whereas our implementation can cope with any query sequence length. This is a major advantage of our method which makes it completely useful in real world bioinformatics applications. Note that the difference between throughput ratios and execution time ratios in Table 3 as due to the use of different versions of the Swiss-Prot database i.e. with different sizes.

Table3. Performance comparison between Munekawa's method [5] and our proposed method

Query length	Execution time T (sec)		Throughput P (MCUPS)	
	Proposed	Munekawa's	Proposed	Munekawa's
63	2.13	2.96	4059	1838
127	4.16	3.38	4189	3244
191	7.36	3.98	3561	4121
255	9.63	4.66	3634	4725
511	25.12	8.19	2792	5388
4095	202	impossible	2782	impossible

Table 4 presents comparative results with another recent GPU implementation of the Smith-Waterman algorithm, from Manavski's et al. targeted at an NVIDIA Geforce 8800GTX GPU [4] which was explained in detail in section 4.A above. This implementation has obvious speed advantages as can be seen from Table 4 thanks to the higher amount of parallelism allowed by computing several alignment matrices in parallel coupled with a simpler synchronization mechanism allowed by the fact that only one thread is associated to each alignment matrix calculation. Nonetheless, this implementation suffers from the local memory size bottleneck which limits the size of query sequences to be processed to 2500 approximately. Our implementation on the other hand does not suffer from any such limitation, and can thus be fully adopted in a real world bioinformatics application.

Table4. Performance comparison between Manavski's method [4] and our proposed method, substitution matrix was used.

Query length	Execution time T (sec)		Throughput P (MCUPS)	
	Proposed	Manavski's	Proposed	Manavski's
63	8	2.98	1080	1849
127	15.5	5.88	1124	1889
255	35	12.31	1000	1811
511	81.8	24.89	857	1795
4095	633.6	impossible	885	impossible

In addition, we have compared the performance of our GPU implementation with a widely used optimised CPU implementation of the Smith-Waterman algorithm, namely SSEARCH from the FASTA set of programs [13]. Table 5 presents comparative results of our GPU implementation with an equivalent SSEARCH (version 35.04) implementation on a Pentium4 3.4GHz desktop computer running Windows XP Professional. This shows that our GPU implementation outperforms an equivalent CPU implementation by up to 15x.

Table5. Performance comparison between SSEARCH and our proposed method

Query length	Execution time T (sec)		Throughput P (MCUPS)	
	Proposed	SSEARCH	Proposed	SSEARCH
63	8	125	1080	70
127	15.5	210	1124	83
255	35	424	1000	83
361	56.4	536	880	92
511	81.8	779	857	90

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel technique for the implementation of the Smith-Waterman algorithm on CUDA-compatible GPUs. The technique solves the query length limitation reported in previous GPU implementation of the Smith-Waterman algorithm, as it can cope with any query or subject sequence sizes. Central to this technique is a divide and conquer approach to alignment matrix calculation in which the size of sub-matrix calculation is dictated by the available computing and memory resources in the GPU hardware, with thread reuse across all sub-matrix calculations. This however comes at a speed overhead due the storing and loading of temporary intermediate data in the global memory. Despite this speed penalty, our GPU implementation still outperforms an optimised CPU-only implementation by up to 15x.

Future work will harness our divide and conquer technique with the single thread per alignment matrix parallelising strategy outlined in section 4.B, and compare the results with the implementation reported in this paper. We also plan to accelerate other biological sequence alignment algorithms on GPUs including the BLAST algorithm and biological sequence analysis using Hidden Markov Models (HMMs).

REFERENCES

[1] R. Durbin, S. Eddy, A. Krogh and G. Mitchison. "Biological Sequnce Analysis: Probabilistic Models for Proteins and Nucleic Acids". Cambridge University Press, Cambridge University UK, 1998.

[2] M. Charalambous, P, Transcoso and A. Stamatakis. "Intial experiences porting a bioinformatics application to a graphics processor". In Processings of 10th Panhellenic Conference on Informatics, 2005.

[3] W. Liu, B. Schmidt, G. Voss, A. Schroder and W. Muller-Wittig. "Bio-Sequence Database Scanning on GPU". In proceeding of 20th IEEE International parallel & distributed processing symposium: 2006 (IPDSP 2006) HICOMB workshop Rhode Island, Greece. 2006.

[4] S. A. Manavski and G. Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment". BMC Bioinformatics 2008, 9(Suppl 2):S10. 2008-3-26

[5] Y. Munekawa, F. Ino and K. Hagihara. "Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU". ISBN: 978-1-4244-2844-1. 2008-12-08

[6] nVIDIA Corporation. "CUDA Programming Guide Version 1.1", http://developer.nvidia.com/cuda/

[7] K. J and W. R. "Linear algebra operators for gpu implementation of numerical algorithm". ACM Trans. Graph, 22:908-916, 2003

[8] P. Agarwal, S. Krishnan, N. Mustafa and S. Venkata-subramanian. "Streaming geometric optimization using graphics hardware". In Proc. 11th European Symposium on Algorithms, 2003.

[9] F. Xu and K. Muller. "Ultra-fast 3d filtered back-projection on commodity graphics hardware". In IEEE International Symposium on Biomedical Imaging'04, 2004.

[10] M. Farrar. "Striped Smith-Waterman speeds database searches six times over other SIMD implementations". Bioinformatics, vol. 23, no. 2, pp. 156–161, Jan. 2007.

[11] T. F. Smith and M. S. Waterman. "Identification of common molecular subsequences". J. Molecular Biology, vol. 147, pp. 195–197, 1981.

[12] A. Bairoch, R. Apweiler. "The SWISS-PROT protein knowledgebase and its supplement TrEMBL". Nucleic Acid Research, release 56.3, 14-Oct-08

[13] W. R. Pearson and D.J. Lipman. "Improved tools for biological sequences comparison". Proc Natl Acad Sci USA 1988, 85:2444-2448