Cluster Comput (2009) 12: 341–352
DOI 10.1007/s10586-009-0089-8

Scalable and highly parallel implementation of Smith-Waterman on graphics processing unit using CUDA

Ali Akoglu · Gregory M. Striemer

Received: 9 September 2008 / Accepted: 8 May 2009 / Published online: 11 June 2009

Abstract Program development environments have enabled graphics processing units (GPUs) to become an attractive high performance computing platform for the scientific community. A commonly posed problem in computational biology is protein database searching for functional similarities. The most accurate algorithm for sequence alignments is Smith-Waterman (SW). However, due to its computational complexity and rapidly increasing database sizes, the process becomes more and more time consuming making cluster based systems more desirable. Therefore, scalable and highly parallel methods are necessary to make SW a viable solution for life science researchers. In this paper we evaluate how SW fits onto the target GPU architecture by exploring ways to map the program architecture on the processor architecture. We develop new techniques to reduce the memory footprint of the application while exploiting the memory hierarchy of the GPU. With this implementation, GSW, we overcome the on chip memory size constraint, achieving 23× speedup compared to a serial implementation. Results show that as the query length increases our speedup almost stays stable indicating the solid scalability of our approach. Additionally this is a first of a kind implementation which purely runs on the GPU instead of a CPU-GPU integrated environment, making our design suitable for porting onto a cluster of GPUs.

A. Akoglu (✉) · G.M. Striemer
Department of Electrical and Computer Engineering, University of Arizona, 1230 E. Speedway Blvd, Tucson, Arizona 85721, USA
e-mail: akoglu@email.arizona.edu

G.M. Striemer
e-mail: gmstrie@email.arizona.edu

Keywords Graphics processing unit · Scalable · Parallel · Alignment · Smith-Waterman · CUDA

1 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 [8, 11, 12]. 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 [4, 8, 11]. 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, 11]. The query sequence is a protein sequence which the user is scoring against a given database. A protein database is a database containing protein sequences with known functionality. 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 sensitivity in finding the best possible local alignments. In this work we map SW onto the G80 NVIDIA 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 GPUs.

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. 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 making SW a practical option for life scientists through scalable high-performance cluster based systems.

The GPU architecture is perfect for algorithms such as SW, where the same program code needs to be executed on several different sets of data. The G80 architecture GPUs run several thousand threads of code concurrently on different sets of data. This is why we have chosen it as our vehicle 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 approach is more desirable for GPU scalability. We believe we have more effectively exploited the GPU architecture to increase the speed of alignments with SW; achieving over 23 times speedup while utilizing only the GPU. The rest of this paper is organized as follows: Sect. 2 describes the GPU and CUDA architectures. Section 3 describes the Smith-Waterman algorithm. Related work is discussed in Sect. 4. Our method of mapping the algorithm on the GPU architecture is detailed in Sect. 5. Section 6 contains our results. Finally, Sect. 7 concludes the paper and contain an outlook for further research of Smith-Waterman on GPUs.

2 GPU/CUDA architecture

Until now it has been much more difficult to program on a GPU, due to reformulating algorithms and data structures using computer graphics primitives (e.g. textures, fragments) such as with OpenGL [5]. CUDA allows the programmer to efficiently program in a manner similar to C to exploit the latest NVIDIA graphics cards. In CUDA, parallelized programs 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, and thread blocks are organized into a grid of blocks. The device schedules blocks for execution on the multiprocessors in the order of their placement [9]. The placement is determined by the user, and is specified when launching the kernel from the host. These containing elements can be one, two, or three dimensional. Figure 1 illustrates the general layout of a grid of thread blocks on the device.

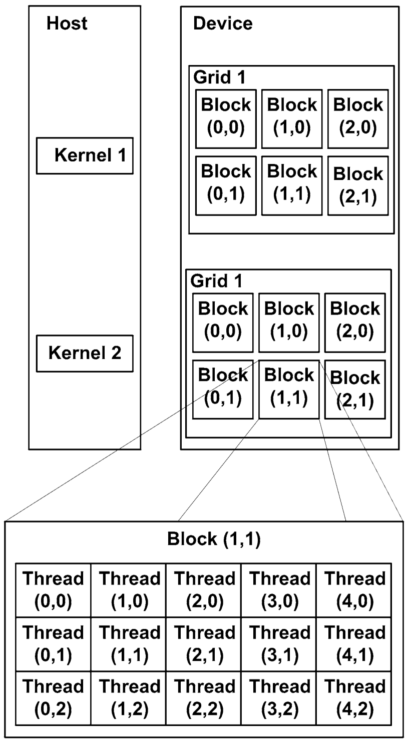

Fig. 1 Structure of CUDA grid blocks [9]

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 [9]. The fastest memories are the shared memories and registers. These can be read from and written to directly by each thread. The shared memory can share data within a block of threads, but is limited to only 16 KB per multiprocessor. The available registers are limited to 8,192 32-bit registers per multiprocessor. 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 8 KB. 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 [9]. The texture cache is designed so that threads reading addresses with close prox-

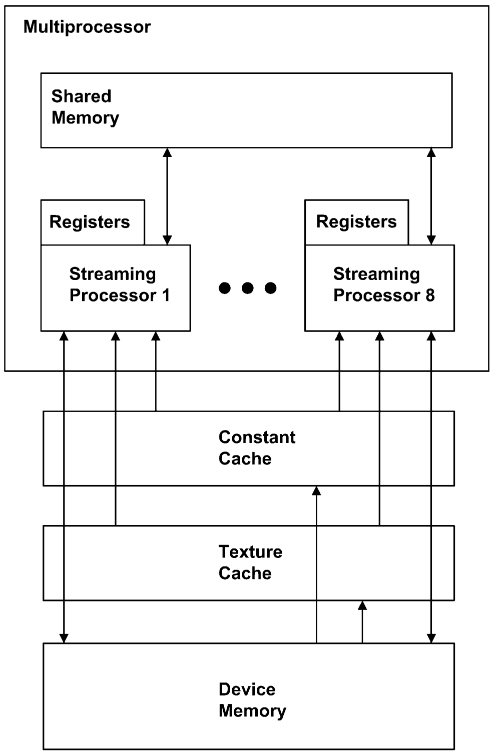

Fig. 2 CUDA hardware model memory layout [9]

imity will achieve better transfer rates. Figure 2 provides a basic memory layout.

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 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. To help programmers more efficiently manage register usage, NVIDIA has made available an occupancy calculator for managing register use in programs as well as efficiency.

3 Smith-Waterman algorithm

The SW algorithm, besides being the most sensitive for searching protein databases for sequence homology, is also the most time consuming [3, 7, 11]. Homology simply refers to similarities due to a common ancestry or descent. SW provides the highest scoring matches between two proteins. It does so by comparing two sequences and computing a score of similarity [5]. This similarity score is sometimes referred to as the SW score. The algorithm is represented with the

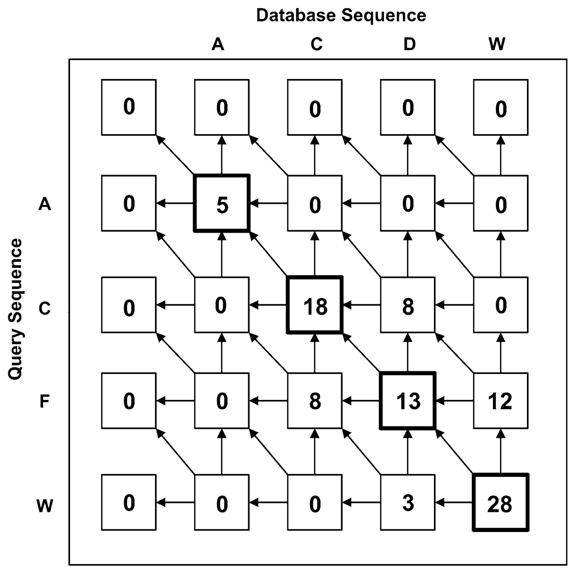

Fig. 3 Smith-Waterman matrix filled with scores. A given matrix cell is dependent on north, north-west and west cells. Each cell is filled in with its respective score assuming the substitution matrix is BLOSUM50

following equation [10]:

$$H_{i,j} = \max\{H_{i-1,j-1} + S_{i,j}, H_{i-1,j} - G, H_{i,j-1} - G, 0\} \tag{1}$$

In this equation i is the position in the query sequence, and j is the position in the database sequence. $S_{i,j}$ 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. There are several different types of substitution matrices available, which have different scoring metrics. In the equation, a zero is added to prevent any cells in the matrix from going negative [1]. Each cell in the matrix created by the two sequences has three data dependencies in the directions of north, north-west, and west. These dependencies are described in the equation by $H_{i-1,j} - G$ for north, $H_{i-1,j-1} + S_{i,j}$ for north-west, and $H_{i,j-1} - G$ for west. G represents a gap penalty. If the maximum score for a given cell is dependent upon the west or north cells, a gap is created thus causing a gap penalty to occur. The provided equation assumes a linear gap penalty, which means that if a gap is extended it will still have the same penalty as a created gap. A gap extension simply means that a given cell created a gap to achieve its score, and the cell from which it is dependent was also created with a gap.

In Fig. 3 we display the dependencies of two sequences being compared with SW as well as the calculation values of each cell in the matrix. The scores were produced using a BLOSUM50 matrix, which is a common substitution matrix for sequence alignments [1]. The outer walls of the

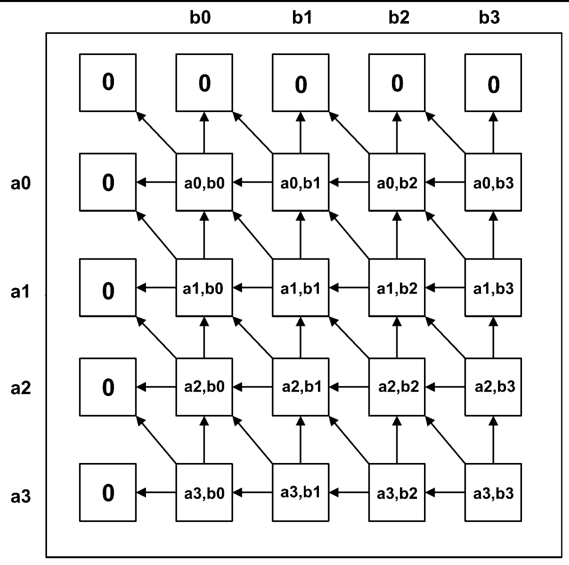

Fig. 4 Data dependencies of Smith-Waterman

```
/*
  The main program takes care of database
  transfers, memory allocation, and
  launching of the kernel on the GPU
*/

main() {
        load (database to host memory from file )

        transfer database (host memory to GPU memory)

        launch kernel

        transfer results(GPU memory to host memory)

        write to file (results on host to text file )
} //end
```

Fig. 5 Pseudo code for top level program flow. The actual SW algorithm is run within the kernel. Pseudo code for the kernel is located in Fig. 10

matrix are padded with zeros. The reasoning for this is so that the boundary cells of the matrix have starting comparison values. In order to comply with the data dependencies of the algorithm, there are three patterns in which cells can be calculated. Sequences can be calculated by row, by column, or by diagonals. Any given row column or diagonal cannot be computed before its preceding row, column, or diagonal. This is due to the aforementioned dependencies. For example, in Fig. 4 if one were to calculate by column, they would calculate cell scores "a_0 to a_3" for b_0, then "a_0 to a_3" for b_1 and so forth. The highest scoring cell in the matrix will contain the SW similarity alignment score for the pair. This score is often viewed as the distance needed to change one sequence into another. Sometimes a trace-back procedure is performed on high scoring sequence pairs, which returns the actual sequence residues that contributed to the similarity score. Our implementation of SW returns the alignment scores, but does not return a trace-back of the two sequences. Thus we do not go into detail of a trace-back procedure.

4 Related work

As of this writing there is only one published attempt to implement SW using CUDA. This is done on an NVIDIA GeForce 8800 GTX GPU by Manavski [7]. They analyze their work against a serial implementation of the algorithm, called SSEARCH, widely used by life sciences community. This program is available from the University of Virginia [13]. They also evaluate their method against an implementation by Farrar, which is optimized for multi-threading [2].

The longest query sequence tested in Manavski's implementation has 567 residues. It is at this length they report a database alignment time of 11.96 s using dual GPUs, and 23.32 s using a single GPU. Manavski reports up to 30 times speedup over the serial implementation SSEARCH. For both single and dual GPU configurations, Manavski utilizes the help of an Intel Quad Core processor. Their method distributes the workload among GPU(s) and the Quad Core processor and the performance results are based on a CPU-GPU integrated environment. Because this is not a pure GPU implementation, their approach raises questions regarding the scalability of the design.

More importantly using the same sequence (567 residues), they report Farrar's multi-threaded implementation taking 20 s on an Intel Quad Core Processor. Therefore, Manavski's design with a single GPU performs worse than Farrar's implementation. They achieve only two times speedup on a platform with 2 GPUs and Quad core processor sharing the workload against a quad core based implementation of Farrar. This indicates that their program architecture is not fitting on to the GPU architecture well. Their implementation has a major drawback in utilizing the texture memory of the GPU that leads to unnecessary cache misses. We summarize the drawbacks of their implementation below.

Manavski's method involves pre-computing a query profile matrix parallel to the query sequence for each possible residue. The purpose of this is to reduce latency caused by looking up similarity scores of sequence pairs. Manavski sorts the database sequences according to length prior to aligning. The sorting is done so all threads within a thread block finish executing at approximately the same time. This implementation has the following issues, which our implementation has overcome as we will describe in the Methods section:

Fig. 6 Detailed look at program flow

```
//Allocate memory on host for protein database
h_db = malloc(size of db);

//Allocate memory on device for database array
d_db = cudaMalloc(size of db);

//Transfer database from file to host
h_db <- db_file;

//Transfer database from host to GPU (Device)
cudaMemcpy(d_db, h_db, size, HostToDevice);

//Allocate memory on host and GPU for results
h_results = malloc(number of sequences);
d_results = cudaMalloc(number of sequences);

// Allocate global GPU memory for temporary values
// needed for calculating SW.
global_temp_cells = cudaMalloc(#sequences x query_len)

//Setup kernel execution parameters
number_threads_per_block = 64;

if(num_sequences % 64 == 0)
num_blocks_per_grid = num_sequences % 64;
else
num_blocks_per_grid = (num_sequences / 64) + 1

//Start Timer
cudaCreateTimer( timer);

//Launch the kernel (performs SW computations)
SW_kernel<<< grid, threads>>>(d_db, d_results);

//Substitution matrix to constant memory
__device__ __constant___   Sbt [26][26] = { };

//Query to constant memory
__device__ __constant___   query = { };

//Transfer results from GPU to host
cudaMemcpy(h_results, d_results, size,
DeviceToHost);

//Stop Timer
cudaStopTimer( timer);

//Write Results to file
results_File <- h_results;

//Free used memory on host and device
free(host variables);
cudaFree(device variables);
```

- 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. This is done to avoid lookup of the substitution values on the internal cycles of the algorithm. With this method, the query profile quickly fills up the texture cache causing major memory delays due to cache misses. Simply, if the query length is larger than size of 356, query profile size becomes larger than 8 KB. As shown in previous section, texture cache size on a GPU is 8 KB. Therefore after 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 to 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 de-

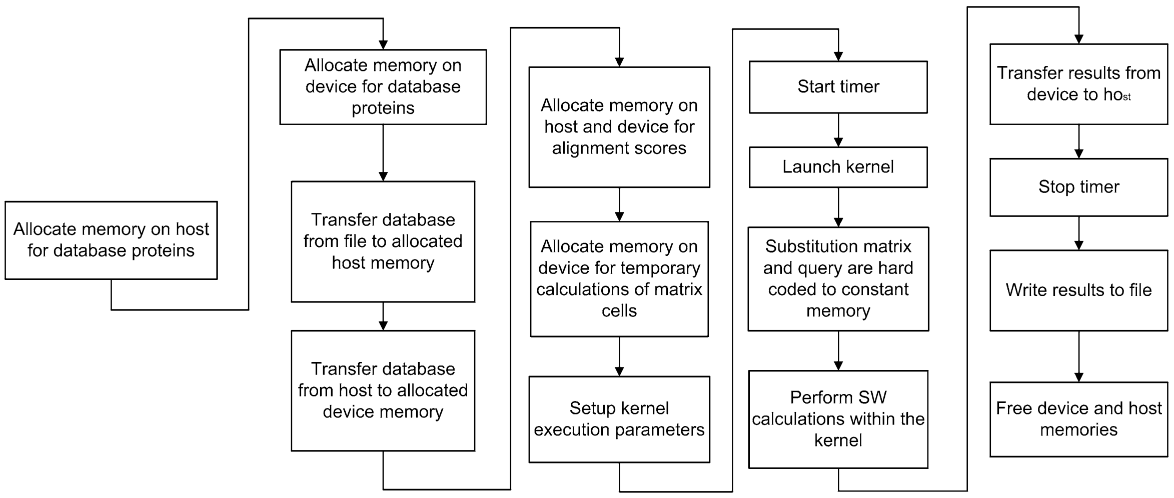
Allocate memory on host for database proteins
Allocate memory on device for database proteins
Transfer database from file to allocated host memory
Transfer database from host to allocated device memory
Allocate memory on host and device for alignment scores
Allocate memory on device for temporary calculations of matrix cells
Setup kernel execution parameters
Start timer
Launch kernel
Substitution matrix and query are hard coded to constant memory
Perform SW calculations within the kernel
Transfer results from device to host
Stop timer
Write results to file
Free device and host memories

Fig. 7 Basic program flow

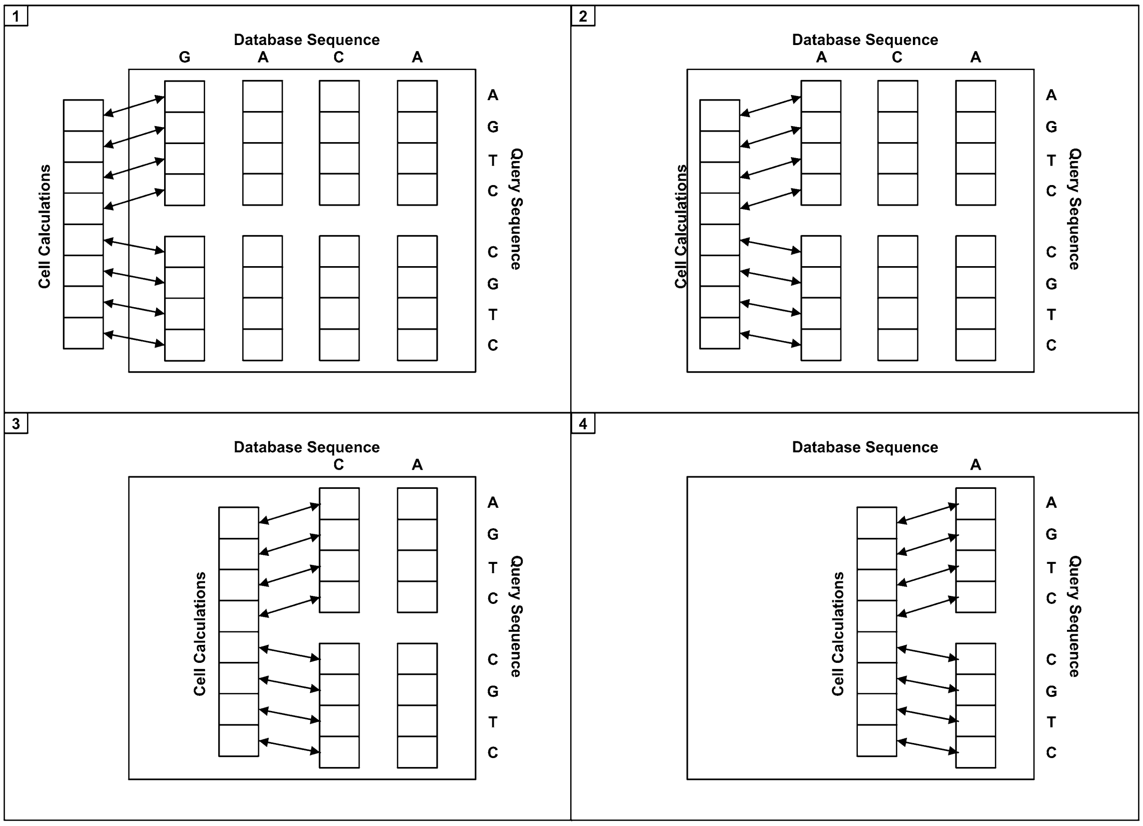

Fig. 8 Example of calculation progression through a 4 residue database sequence aligned with an 8 residue query sequence. The cell calculation blocks are used to store temporary calculation values needed for data dependencies for each column and are stored in global memory. The database sequences are stored in global memory

sign 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 query sequences reach over 400 residues in length, or "serious problems could be encountered". This CPU dependency destroys the chances for GPU cluster scalability in their implementation.

We conclude that Manavski's approach is highly CPU dependent and does not exploit the GPU architecture, because

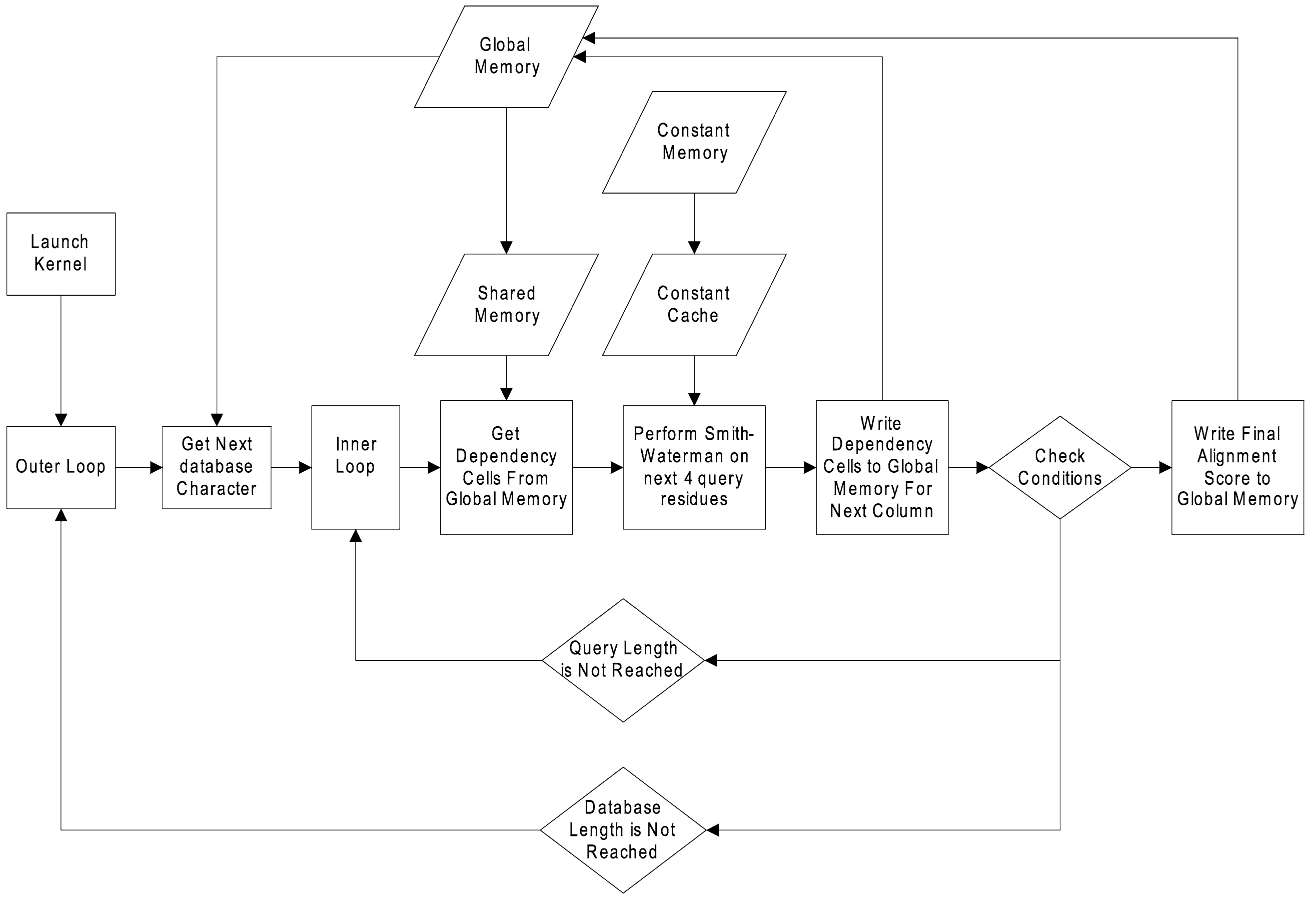
Global Memory
Constant Memory
Launch Kernel
Shared Memory
Constant Cache
Outer Loop
Get Next database Character
Inner Loop
Get Dependency Cells From Global Memory
Perform Smith-Waterman on next 4 query residues
Write Dependency Cells to Global Memory For Next Column
Check Conditions
Write Final Alignment Score to Global Memory
Query Length is Not Reached
Database Length is Not Reached

Fig. 9 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 at a time

their program architecture does not overlap with the hardware architecture successfully. In the next section we introduce a new technique that overcomes the drawbacks of Manavski's approach. We show that our GSW is a scalable, pure GPU implementation with much better performance. We believe that all these features of our algorithm results with execution performance needed by the life sciences community and make it practical for deploying in multi GPU environment.

5 Methods

Unlike the Manavski method, we in no way use the CPU for SW computations. We leave this strictly to the GPU, in order to make our implementation more scalable for cluster based GPU situations. In Figs. 5 and 7 we show the top level pseudo code and basic flow of our program. Pseudo code for the kernel can be found in Fig. 10, with a supplementing flow chart in Fig. 9. The kernel, which runs the actual SW alignments is launched on line 27 of Fig. 5. 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, 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. On lines 30 and 33 of Fig. 5, we place the query sequence and substitution matrix into the constant memory.

Manavski was also only launching 450 thread-blocks at a time on a grid. This means that several kernel launches were necessary, which created additional latency. Since they were running 64 threads per block, they could only process 28,800 alignments per kernel launch. CUDA limits the number of blocks in a grid to 65,535 blocks in a single dimension, which is far above the threshold they were using. This number can get even higher if multiple dimensions are utilized. As shown on the lines proceeding 20 of the pseudo code, we determine our block size by the total number of sequences located in the database. We used 64 threads per

Fig. 10 Kernel pseudo code

```
/*
The kernel is run on each thread . Each thread runs SW on a different database sequence
in parallel . The kernel consists of an inner and outer loop . The outer loop moves through
characters of the database sequence , while the inner loop does the actual SW algorithm
calculations . In the inner loop , four blocks of the matrix are calculated at a time by
column. The number of blocks calculated is equal to the length of the query sequence .
*/
/*_____________ Begin Outer Loop _____________ */
for (each database sequence character ) {
    Get characters from database
    /*_____________ Begin Inner Loop _____________ */
    for (each query sequence character ) {
        for(each of 4 query character sub -blocks ) {
            find maximum value of {north_west_blockl + Si,j,
                                    north _block−gap,
                                    west_block – gap,
                                                0}
            keep track of maximum score
        }
    }
} //End of Outer Loop
```

block just as Manavski. However, by setting up the execution parameters as we have for the kernel, we are able to do all alignments with a single kernel launch. CUDA offers a built in timer function, which we have utilized to assess the performance of the alignments on the kernel. The timer is referred to in lines 25 and 40 of the pseudo code.

In our implementation we calculated the SW score from the query sequence and database sequences by means of columns, four cells at a time similar to Manavski. The progression of these calculations can be seen in Fig. 8 for a database sequence of length 4 and a query of length 8. When cells are calculated, their updated values are placed in a temporary location in the global memory. This is represented by "cell calculations" in the figure. 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.

The highest SW score is simply kept track of by a single register in the kernel for each thread, and is updated on each pass of the inner loop. After the alignment is complete, the score is written to the global memory as shown in Fig. 8. The entire matrix is not stored in memory, but rather just the temporary cell calculations column. The progression of reading and writing these temporary cell values to the global memory can be seen in Fig. 8, where we provide a flowchart of the kernel. Four dependency values are read at the beginning of the inner loop, and new values for which the next column will be dependent on are written at the end of the inner loop. The dependency cells are initially filled with zeros, to account for the padding around the matrix as described in Sect. 3. The number of dependency cells needed for each alignment is simply equal to the length of the query sequence. This allows us to calculate by scores column by column. Currently our query lengths are limited to 1024 residues, but we are working on some indexing strategies which will allow us 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.

To make access to the substitution matrix easy, we have placed it in the constant memory. One simple yet important obstacle we had to overcome was efficiently accessing the substitution matrix during the internal cycle of the algorithm. Using the modulus operator is extremely inefficient on CUDA [9], so a more traditional hashing function could not be efficiently utilized for accessing the substitution values. Since the substitution matrices are symmetric, and the order of comparison does not matter. Common matrices utilize 23 letters of the alphabet as residues. We took this information and designed a simple yet extremely efficient cost function for accessing substitution matrices.

There are two changes which need to be made to a given substitution matrix in order for our cost function to work. The substitution matrix needs to be re-arranged in alphabetical order, and null characters must be inserted where there

```
/*
4 cells of the SW alignment matrix are calculated on each pass of
the inner loop of the kernel . When the inner loop completes , the
next character is loaded from the database sequence on each pass of
the outer loop . This repeats column by column until the entire
length of the database is completed 4 cells at a time. When the end
of the database sequence is reached by the outer loop , the final
alignment score is recorded . This code is for an individual thread ,
and each thread works on a different database sequence . Only 4
cells are worked on at a time , due to the limitations of the shared
memory. The global memory cells are set to zero initially by the
main CUDA program outside of the Kernel . This accounts for the
padding of zeros around the border of the matrix . S(i,j) (The
substitution matrix ) and the query sequence are located in the
constant memory .
*/
/*______________ Begin Outer Loop ______________ */
for(i = 0; i < database_Sequence_length; i++)
//Get characters from database 1 at a time
database_character[i] = Device_Database[i];

//Reset north and north -west cells to zero
north_cell      = 0;
north_west_cell = 0;
//temp1 and temp2 are for temporary cell calculations of SW
score_temp1 = 0;
score_temp2 = 0;

/*______________ Begin Inner Loop ______________ */
for (j = 0; j < query_length , j++)
// Get 4 cell values from global memory
// store them in shared memory . Total number of
// cells is same as length of query sequence .
// The inner loop is repeated for entire length
// of query sequence for each database sequence
// character
shared_cell_1 = global _cell_1;
shared _cell_2 = global _cell_2;
shared _cell_3 = global _cell_3;
shared _cell_4 = global _cell_4;
/*____________ Calculate New Cell 1____________ */
//Perform Smith-Waterman on cell
//S(i,j) is the score from the substitution matrix
score_temp1 = max(0, north_west_cell + S(i,j) );
score_temp2 = max(north_cell - gap, score_temp1);
shared_cell_1 = max(shared _cell_1 - gap, score_temp2);
//Set new Score
score = max(score, score_temp2);
//set a new north value
north_cell = score_temp2;
//set a new north _west value
north_west_cell = shared _cell_1;

/* ____________ Calculate New Cell 2____________ */
//Perform Smith-Waterman on cell
score_temp1 = max(0, north _west_cell + S(i,j) );
score_temp2 = max(north_cell - gap, score_temp1);
shared_cell_2 = max(shared _cell_2 - gap, score_temp2);
//Set new Score
score = max(score, score_temp2);
//set a new north value
north _cell = score_temp2;
//set a new north _west value
north _west_cell = shared _cell_2;

/* ____________ Calculate New Cell 3____________ */
//Perform Smith-Waterman on cell
score_temp1 = max(0, north _west_cell + S(i,j) );
score_temp2 = max(north_cell - gap, score_temp1);
shared _cell_3 = max(shared _cell_3 - gap, score_temp2);
//Set new Score
score = max(score, score_temp2);
//set a new north value
north _cell = score_temp2;
//set a new north _west value
north _west_cell = shared _cell_3;

/* ____________ Calculate New Cell 4____________ */
//Perform Smith-Waterman on cell
score_temp1 = max(0, north _west_cell + S(i,j) );
score_temp2 = max(north_cell - gap, score_temp1);
shared _cell_4 = max(shared _cell_4 - gap, score_temp2);
//Set new Score
score = max(score, score_temp2);
//set a new north value
north _cell = score_temp2;
//set a new north _west value
north _west_cell = shared _cell_4;

/*___Record updated Cell Scores to Global Memory ____*/
global _cell_1 = shared _cell_1;
global _cell_2 = shared _cell_2;
global _cell_3 = shared _cell_3;
global _cell_4 = shared _cell_4;
//End of inner Loop
end for
//End of Outer Loop
end for
/*______ Write Final Scores to Global Memory ______*/
global _SW_Results = score;
```

Fig. 11 More in depth look at SW execution in the kernel

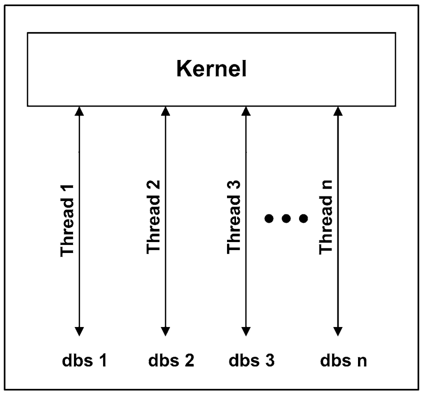

Fig. 12 SW parallelization on the GPU architecture. dbs is a given database sequence being aligned and Thread is the current thread launched from the kernel. Each Thread aligns a different database sequence

Table 1 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

are unused characters of the alphabet. When using BLOSUM matrices, there are only 3 unused characters which must be added. The cost function can be seen in Eq. 2.

$$S_{i,j} = (\text{ascii}(S_1) - 65, \text{ascii}(S_2) - 65) \tag{2}$$

In Eq. 2, S_1 is a residue from the query sequence and S_2 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 which we are looking for very efficiently. Table 1 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.

For example: if S_1 is B, and S_2 is D, then we will get the following from Eq. 2, using Table 1 for substitution values:

$$\begin{aligned} S_{i,j} &= (\text{ascii(B)} - 65, \text{ascii(D)} - 65) \\ &= (66 - 65, 68 - 65) \\ &= (1, 3) \\ &= -3 \end{aligned} \tag{3}$$

Since the substitution matrix is symmetric, the order of S_1 and S_2 does not matter. The equation allows us to take up a relatively small amount of space in the constant memory cache, while at the same time having an efficient way to access the values of the substitution matrix as needed within the kernel. This is how we take care of accessing the substitution matrix rather than creating a query profile such as Manavski. Similar to the Manavski implementation we have used an ordered database according to length for our searches. We have done this for the same reasons, as previously mentioned.

By utilizing the GSW method we are able to reduce the time complexity of SW from $O(nm)$ to $O(mn/t)$. M and n are the lengths of the sequences being aligned and t is the number of threads launched from the kernel at any given time on the GPUs multiprocessors. The overall space complexity of GSW on the GPU is $(D_n + QD_p)$, where D_p is the number of proteins in the given database, Q is the query sequence length, and D_n is the number residues in the database.

Figure 12 provides a simple example of how SW has been parallelized for the GPU architecture. Each thread launched from the kernel runs the algorithm on a different database sequence. GSW can launch 512 threads at once on each of the 16 multi-processors, for a total of 8,192 different SW alignments happening at once.

6 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:

- Speedup gained from the GPU side is not measurable: Implementation is not a true GPU mapping as it relies on CPU computations
- 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 GSW and compared it against SSEARCH. Our SSEARCH tests were performed on a Gateway E-6300 containing a 3.2 GHz Pentium 4 processor, and

Table 2 GPU vs SSEARCH Alignment Results. Database used: Swissprot (Aug 2008) 392,768 sequences. Note: SSEARCH runs on 3.2 GHz processor, GPU runs at 1.35 GHz

Protein ID	Protein length	GPU time (s)	SSEARCH time (s)	Speedup	Clock cycles SSEARCH (billions)	Clock cycles GPU (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	377.55	15.32

1 GB RAM. The Gateway system is running Windows Vista. Since SSEARCH will perform differently depending on the host system's processor, 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. The only item that is changing is the query sequence, to test the change in lengths on the algorithms performance. SSEARCH only takes two inputs from the user, a file with the query sequence and the database to be searched. The program is set to run with a BLOSUM50 substitution matrix, a gap penalty of -10, and a gap extension penalty of -2. These settings cannot be changed, so setting the gap extension penalty to -10 to make it linear could not be done for these tests. To accommodate to their settings, we have used a BLOSUM50 substitution matrix for our tests. We have set up our implementation with a gap penalty of -10. Any gap extensions in our implementation will also have a penalty of -10.

GSW was tested against SSEARCH using query sequence lengths ranging from 4 to 1024 residues. Table 2 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. Recall, 1.35 GHz is the clock speed of a streaming processor on the architecture. This allowed us to get more reasonable look at performance based on cycles to compute the alignments rather than, just the execution speed which is tied to the speed of the processor. On average GSW 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.

Another significant observation we have made out of the experimental studies is that as the query length increases our speedup based on "cycles ratio" metric almost stays stable, indicating the solid scalability of our approach.

The speedup for GSW can be represented as Eq. 4:

$$\text{Speedup} = \frac{\text{SSEARCHExecutionTime} * \text{CPCS}}{\text{GSWExecutionTime} * \text{GPCS}} \tag{4}$$

In this equation, CPCS is the clock speed of the PCs processor, and GPCS is the clock speed of the GPUs processor. By finding the ratio of total clock cycles for execution between the two implementations, we are able to establish a speedup which is not dependent on the clock speed.

7 Conclusion and future work

Through this work we have found that SW can be effectively mapped to the GPU architecture using CUDA. We were able to achieve up to 23 times better performance over the serial method SSEARCH, and have designed GSW in such a way as to allow massive scalability. Since GSW only relies on the GPU for computations, we will be able to achieve unprecedented speedup on GPU clusters. A database can simply be split evenly among GPUs, and each GPU can work independently on alignments. With a simple brute-force strategy, we can assume there is two times speedup for every doubling of GPU resources. Therefore we expect to observe 40 times speed up range when 2 GPUs are employed in our system while Manavski's CPU-GPU platform achieves a max of 30 times speed up. With current NVIDIA GPU server clusters, this is a very real and exciting possibility which will be explored.

In the future we plan to exploit the GPUs scalability extensively, by creating methods to form cluster configurations for the GPU and SW. We are also planning optimizations to allow the user to utilize longer query sequences, as well as different gap penalty configurations other than linear penalties. We will also look into ways to increase the number of threads in each block from 64 to 96.

Acknowledgements We would like to thank Nirav Merchant and Susan J. Miller from the Arizona Research Labs for their input and guidance on Smith-Waterman algorithm and its applications.We would also like to thank Yang Song for his contributions to the paper.

References

1. Eddy, S.R.: Where did the BLOSUM62 alignment score matrix come from? Nat. Biotechnol. **22**, 1035–1036 (2004)
2. Farrar, M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics **23**, 156–161 (2007)
3. Walker, J.M.: The Proteomics Protocols Handbook. Humana Press, Clifton (2005), pp. 504
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 1–5, 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 **9**(Suppl 2), S10 (2008)
8. Mount, D.W.: Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, Cold Spring Harbor (2004), pp. 64–65, 71–87
9. NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide. http://www.nvidia.com/object/cuda_develop.html (2008)
10. NVIDIA Corporation: NVIDIA Tesla C870 overview. http://www.nvidia.com/object/tesla_c870.html (May 2008)
11. Sanchez, F., Salamí, E., Ramierez, A., Valero, M.: Performance analysis of sequence alignment applications. In: Proceedings of the IEEE International Symposium on Workload Characterization, pp. 51–60 (2006)
12. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. **147**, 195–197 (1981)
13. UVA: FASTA program webpage. http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml (2008)

Ali Akoglu is an assistant professor in the Department of Electrical and Computer Engineering at the University of Arizona and the director of the Reconfigurable Computing Lab. He received his Ph.D. degree in Computer Science from Arizona State University in 2005. His research interests lie in the fields of reconfigurable architectures, high performance scientific computing and CAD tools for FPGA design and application specific instruction set processor design. His recent interests include developing high performance floating point arithmetic core for reconfigurable systems, FPGA based built in self testing and self healing for deep spacecraft missions and coarse grain reconfigurable architecture tailored to video compression applications.

Gregory M. Striemer is a M.S. student in the Department of Electrical Computing Engineering at the University of Arizona. He currently does research in the Reconfigurable Computing Lab at the University of Arizona, mapping high performance scientific computing applications to GPU architectures. His current interests include the mapping of scientific applications to multi-core architectures, as well as high performance architecture design. He received his B.S. in Business Management in 2006 in Phoenix, Arizona from the University of Phoenix.