AN EFFICIENT IMPLEMENTATION OF SMITH WATERMAN ALGORITHM ON GPU USING CUDA, FOR MASSIVELY PARALLEL SCANNING OF SEQUENCE DATABASES

Łukasz Ligowski, Witold Rudnicki
Interdisciplinary Centre for Mathematical and Computational Modelling
University of Warsaw
Warsaw, Poland
e-mail: W.Rudnicki@icm.edu.pl

Abstract— The Smith Waterman algorithm for sequence alignment is one of the main tools of bioinformatics. It is used for sequence similarity searches and alignment of similar sequences. The high end Graphical Processing Unit (GPU), used for processing graphics on desktop computers, deliver computational capabilities exceeding those of CPUs by an order of magnitude. Recently these capabilities became accessible for general purpose computations thanks to CUDA programming environment on Nvidia GPUs and ATI Stream Computing environment on ATI GPUs. Here we present an efficient implementation of the Smith Waterman algorithm on the Nvidia GPU. The algorithm achieves more than 3.5 times higher per core performance than previously published implementation of the Smith Waterman algorithm on GPU, reaching more than 70% of theoretical hardware performance. The differences between current and earlier approaches are described showing the example for writing efficient code on GPU.

Keywords- sequence alignment; GPGPU; CUDA; Smith Waterman;

I. INTRODUCTION

In recent years the traditional method of improving the performance of CPUs, namely by increasing the clock frequency has exhausted its potential and performance growth is achieved by increasing the number of computing cores and the size of the on-chip cache. Current top of the line traditional CPUs are built with four cores and eight core designs are already in development. In the near future one can expect designs with more than 30 cores. In the seminal report Berkeley team [1] argues that future computers will be built around processors with hundreds, or even thousands of cores.

One should note that chips with comparable number of computing cores are already present on the market in large quantities. Current generation of graphic cards, such as Nvidia GeForce or ATI Radeon contain hundreds of computing cores. The peak performance of these cards is at least one order of magnitude higher than that of traditional CPUs and they and are capable of performing large scale computations for all problems where data parallel approach is feasible. Algorithms for several suitable problems have been already implemented for these platforms; in many cases showing a very good performance. The examples from diverse disciplines, such as quantum chemistry [2, 3], computational fluid dynamics [4-6], astrophysics [7], computer science [8] as well as search of similar sequences of biological macromolecules [9, 10] have been recently reported. All these examples fall into categories, which have been mentioned in the report of the Berkeley group as most challenging for the new massively parallel paradigm for software development. Therefore one may argue, that General Purpose GPU computing is the first step towards this new paradigm.

Programs for searching similarity between biological macromolecules are ubiquitous tools of molecular biology. The most common approach is based on representing molecules as strings of symbols and finding statistically important matches between those strings. The score of the match is obtained using the so-called amino acid similarity matrices (AASM) [11, 12]. The program cited most often, BLAST [13, 14], performs an alignment of a query molecule against all molecules in a database. It uses an approximate heuristic algorithm which is about two orders of magnitude faster than exact Smith and Waterman algorithm (SW) [15, 16]. Nevertheless, SW algorithm is widely used in many bioinformatical applications, either as the last stage of sequence similarity search performed with approximate algorithms [17, 18], or within more advanced algorithms, such as profile-profile methods [19]. Algorithms for sequence matching are also used in the so-called next generation sequencing methods [20, 21], which are used for rapid sequencing of whole genomes. These methods generate millions of relatively short sequences, which then need to be assembled. It has been shown that application of the exact SW algorithm as a part of the assembly algorithm significantly improves the quality of assembled sequence data [22].

Over the years there were several attempts to improve execution speed of the SW algorithm. In particular vector instructions of modern scalar processors were used for improving speed of execution of a single alignment of two sequences [23-26]. An alternative idea is to execute alignment of several database sequences with a single query [27, 28]. In these implementations the speed increase was achieved for a simplified version of the algorithm, which does not store information required for alignment reconstruction, and uses one byte integers for scores. This algorithm is used to detect scores above certain limit and reject non-similar sequences. Then the hits are processed

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

again with a full version of the algorithm which finds correct alignments. The overall speedup is achieved due to the fact, that only a very small fraction of sequences in a database is similar to the query and need to be processed again.

Recently parallel versions of the simplified SW algorithms have been implemented on Nvidia GPU, first using graphic library [29] and, after introduction of CUDA environment for GPU programming [30], also on this platform [10]. The CUDA implementation is using two byte integers, which is sufficient for all practical applications and its performance does not depend on the similarities between sequences. Moreover, changing the representation of numbers to four bytes and including the backtracking information is possible in CUDA, with much smaller decrease in performance than in vectorised versions. Therefore GPU implementation of SW could be used in most applications of SW algorithm. The comparison of the theoretical peak performance of Intel CPU, Cell processor and GPU shows, that the GPU version should be faster than vectorised implementations on other platforms. Unfortunately, the implementation of Manavski and Valle, while delivering good performance, does not fully exploit the potential of the hardware. The highest up to date performance of SW algorithm was reported for parallel implementation on Cell processor [28], with the peak performance on a single CPU reaching 9 GCUPS (giga cell updates per second). The result reported for GPU implementation was 1.7 GCUPS on single GPU.

In the current paper we present a significantly improved implementation of the SW algorithm on GPU using CUDA toolkit. The differences between the two implementations are discussed and suggestions for development of efficient codes are proposed.

II. METHODS

A. CUDA Programming Model

Implementation of the SW algorithm was performed using CUDA library on nvidia GPU. CUDA is a library/extension of C/C++ programming language [30]. It gives a programmer relatively deep control of hardware within a familiar environment. Programming for CPU in a high level language isolates programmer from the hardware details. Most of the CPU transistors are used for this purpose. Programming on GPU is more demanding on a programmer. An efficient implementation of an algorithm on GPU requires in-depth understanding of the device architecture and constraints.

Our implementation of the SW algorithm was developed for and tested on GeForce 9800 GX2 dual core card. A single core of GeForce 9800 GX2 is almost identical with the core of the first generation of CUDA enabled cards. Newer generation of cards share the basic design, with increased number of processing units. There are 128 processing units in a single core of a G92 processor used in the card. Eight processor units, along with 2 special function units are grouped in one Multiprocessor (MP). There is only one instruction issuing unit in a single multiprocessor, therefore all processors perform identical operations on different data. A 16KB on chip cache is dedicated for single MP. Multiprocessor has hardware support for multiple threads. It can execute up to 768 threads concurrently (1024 on newer cards). The bandwidth to the main memory is high (almost 100 GB/s) but the latency is about 500 cycles. High number of threads is used for hiding the high latency of the main memory access. Once the thread requests memory access it can be put to rest waiting for memory, while another threads are executed. On the other hand using shared memory is as efficient as using registers, provided that a right access pattern is employed [30]. Therefore the effective method for programming is based on the following principle: copy the required data from global memory to shared memory, perform as many operations as possible using shared memory write the results to the global memory.

The threads are organised in a structured hierarchical way. The main unit is a block of threads, which for efficiency reasons should be a multiply of and not less than 64. All threads from single block are executed on the same multiprocessor. It is recommended to use at least 192 or 256 threads in order to hide the latency of the main memory. Within a block the threads are organised in warps, each consisting of 32 threads. All threads in the same warp are executed concurrently.

B. Algorithm

Finding similar sequences in the database is performed by string matching. Macromolecular sequences are represented as strings, composed either of 4 or 20 characters for nucleic acids and proteins, respectively. In both sequences the algorithm finds a pair of substrings, which have maximal similarity. The similarity function is given in the form of similarity matrix, such as BLOSUM62, [12] PAM250 [11], for proteins or identity matrix for nucleic acids. Amino acid similarity matrices are based on the statistics of observed mutations of amino acids in the biologically related proteins. The score for aligning two amino acids is highly positive for identical ones. It is positive if mutations involving exchanging one by another are observed more often than by pure chance and it is negative when such mutations are observed less often. The alignment may contain gaps in both sequences, but the introduction of a gap to the alignment is penalised. The penalty is usually a sum of a constant opening penalty and a gap extension penalty which is proportional to the length of the gap. The optimal alignment is defined as the alignment with highest possible score. It cannot be improved neither by elongation nor by trimming.

The Smith Waterman algorithm for finding optimal local alignment is based on dynamic programming approach. The $M \times N$ matrix $\mathbf{A}$ is constructed, where M and N are lengths of two sequences. A matrix element $A_{i,j}$ is filled by a score for aligning *i-th* amino acid in the first protein with *j-th* one in the second protein. Then the problem of finding optimal local alignment is defined as finding a path connecting any starting point (i_0, j_0) in the matrix with any point (i, j), $i > i_0, j > j_0$, with the highest sum of scores

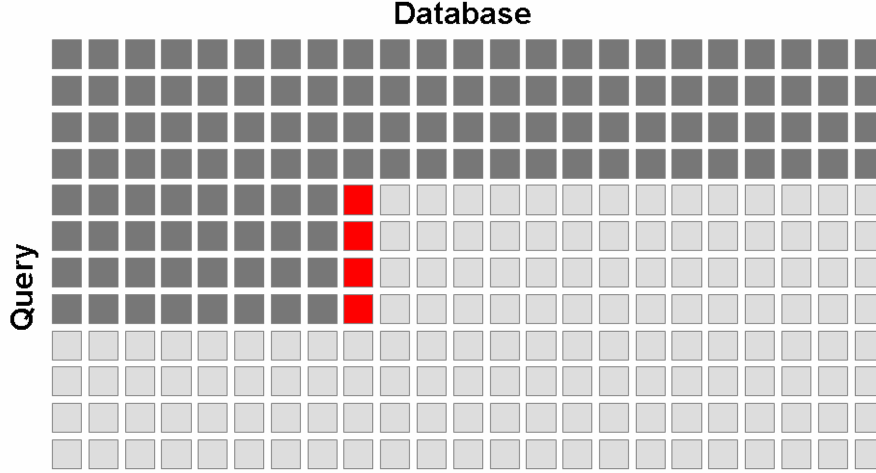


Figure 1. Processing od the dynamic programming matrix. It is processed in horizontal bands. The cells which have been processed are represented in dark gray, cells which are currently processed are represented in red the cells and remaining cells are light gray.

along this path. The perfect alignment without gaps would be a path along single diagonal in the matrix. Alignment with gaps may contain vertical and horizontal segments. To this end the matrix is processed, starting from the upper left corner using the following formula

$$H_{i,j} = \max \begin{Bmatrix} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + A_{i,j} \end{Bmatrix} \qquad (1)$$

where $H_{i,j}$ denotes processed matrix element, $H_{i-1,j-1}$ is the score of the best alignment terminated at $(i-1, j-1)$, $E_{i,j}$ is a score for an alignment with gap along column terminated at (i, j) and $F_{i,j}$ is a score for an alignment with gap along row terminated at (i, j) and $E_{i,j}$ and $F_{i,j}$ are computed earlier using the following formulas

$$\begin{aligned} E_{i,j} &= \max \begin{Bmatrix} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{open} \end{Bmatrix} \\ F_{i,j} &= \max \begin{Bmatrix} F_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{open} \end{Bmatrix} \end{aligned} \qquad (2)$$

where G_{ext} and G_{open} are penalty for extending and opening gap, respectively.

It is easily seen, that all cells, which are in the left upper corner with respect to the current cell, must be already processed. Each cell update requires reading 6 values ($H_{i-1,j-1}$, $H_{i,j-1}$, $H_{i-1,j}$, $E_{i,j-1}$, $F_{i-1,j}$, $A_{i,j}$), writing 3 values ($H_{i,j}$, $E_{i,j}$, $F_{i,j}$) and performing five additions and five calls to function max() (it takes two arguments, hence evaluation of the formula (1) requires three function calls). One additional call to max() is required for checking if the current value of processed matrix is the maximal value. In practical implementations some additional operations may be required for indexing. The main loop of the SW algorithm requires performing 11 arithmetic operations, reading 6 values form the memory and writing 3 values back to the memory. GPU uses 4 bytes integers, therefore a single step of the straightforward implementation involves reading and writing 36 bytes.

Maximal theoretical performance of a single core of the GeForce 9800 GX2 card is 192 billion integer operations per second. The memory bandwidth is 64 GB/s. Maximal theoretical performance of the SW algorithm is therefore 192/11 = 17.3 GCUPS (giga cell updates per second) per core, if limited by operation count (without bookkeeping operations). On the other hand the global memory bandwidth limits performance of the naïve implementation to 64/36 = 1.8 CUPS. It is clearly seen that efficient algorithm needs to be written in such a way that minimizes communication with a main memory.

In practice a small number of additional addressing and bookkeeping operations may be required, decreasing the maximal performance. Their number depends on the implementation details.

One should note that backtracking information for alignment reconstruction is not preserved in this algorithm. Preserving it significantly increases the number of arithmetical and memory operations. When the algorithm is used for scanning database of mostly non-similar sequences, the hits are found in a small fraction of runs. Therefore, it is more efficient to discard the backtracking information for all sequences and repeat the computation with a more costly version including backtracking information for significant hits only.

C. Implementation

The program is executed concurrently on CPU and GPU. The control loop resides on CPU. Program reads pre-formatted database to the GPU memory, and waits for queries. The database is sorted according to size (starting with the longest sequences) and organized in blocks consisting of 256 sequences.

Once the query is received it executes the CUDA kernels which perform database scan with the query. Kernels return the highest alignment score for each sequence. The main program selects the sequences with the score above threshold, which are realigned on the CPU using the algorithm with enabled backtracking. The result is output in the standard BLAST format. Program can utilize either one or both cores.

Each thread performs a complete processing of single pair of sequences. The query sequence is shared by all threads. Since all threads are synchronized the length of the database sequences processed by a single block should be similar (preferably identical). Otherwise all threads would be waiting for the one performing alignment of the longest database sequence.

```
// All matrices are located either in
// global memory (suffix Global) or in
// shared memory (suffix Shared). Other
// variables are located in registers

H_up=GlobalH[j];// read from global memory
F_up=GlobalH[j];//
H_upleft=H_init;// register operation
For (k=0;k<12;k++) do
  // read similarity score from the
  // shared memory
  A = Similarity[query[i],db[j]];
  // read H and E from previous sweep
  H_left=Shared_H[k]; // shared memory
  E_left=Shared_E[k]; // shared memory
  // Compute auxiliary variables
  E=max(E_left-Gext,H_left-Gopen);
  F = max(F_up-Gext,H_up-Gopen);
  // Compute H
  H = max(0,E);
  H = max(H,F);
  H = max(H,H_upleft+A);
  // if this is a first step store H_up
  // in a register for initializing next
  // column
  If (k==0) H_init = H_up;
  // initialize variables for the
  // next step
  Shared_H[k]=H;
  Shared_E[k]=E;
  H_upleft = H_left;
  H_up = H;
  F_up = F;
Done
// Write variables to global memory for
// next sweep
Global_H[j]=H;
Global_F[j]=F;
```

Figure 2 Pseudocode for the inner loop of the program

The dynamic programming matrix is processed horizontally. In each sweep 12 cells high band is processed, see Figure 1**.** The main loop is executed for 12 cells columns of this band. Slow global memory is accessed only at the initialization and termination of the loop. All operations within the loop are performed in fast shared memory and registers. The width of the band is limited by availability of shared memory.

The pseudocode for the computing loop in the individual thread is displayed in Figure 2. There are only 2 global memory transactions per 12 steps. Additionally $H_{i,j}$ and $F_{i,j}$ are stored as half-word integers in the single integer variable. Therefore the total required bandwidth is 8 bytes per 12 cell updates. A detailed representation of memory operations is displayed in Figure 3. Therefore, due to efficient usage of shared memory we managed to raise the memory constraint of the theoretical performance at

$$64 \text{ GB/s} / 8B / 12 \text{ Cell Updates} = 96 \text{ GCUPS}.$$

Taking into account that computational capabilities of the core limit performance at 17 GCUPS we may conclude that our implementation is not limited by memory bandwidth.

Each kernel has 16 blocks in each kernel, and a single block consists of 256 threads – each thread processes a single sequence. The choice of 4096 threads per kernel is dictated partially by limitations of architecture and partially by optimization of performance. There are 16 multiprocessors in each core, each executes a single block of threads. The number of 256 threads per block is limited by the number of registers (8192 per multiprocessor). The SW main routine needs 29 registers, and therefore the highest multiply of 64 that can be concurrently executed is 256. In the current 'proof of concept' version of the code all threads perform identical operations. It is preferable to minimize the number of sequences executed by a single kernel, to minimize length differences within a single kernel. Altogether a single kernel performs comparison of the single query with 4096 database sequences. This configuration gives the optimal results on the 9800 GX2 card.

D. Tests

Tests were performed on the SwissProt sequence database, release 56.5 of 25-Nov-08 from Swiss Institute of Bioinformatics. Due to some technical limitations of the current implementation the practical limit for the database sequence length is slightly less than 1000 amino acids. This is a limit of the 'proof of concept' version of the code, which will be removed in further development. The subset of SwissProt contains 388517 proteins (124041327 residues - 85% of total database length). The database was sorted according to size. Thirty seven sequences were randomly chosen from the database and used as a query. The lengths of the selected queries varied between 10 and 1000 residues, with coverage density decreasing with the length. Each query

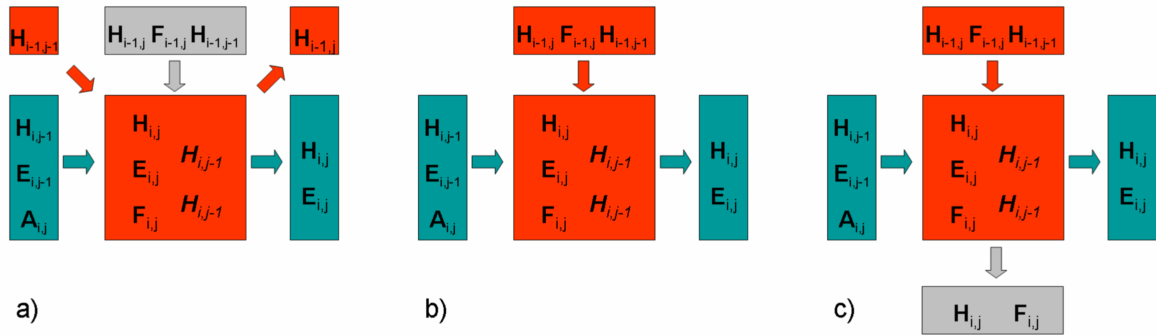


Figure 3. Memory operations in the inner loop of the algorithm at three stages, (a) initialisation, (b) continuation and (c) termination. The global memory, shared memory and registers are represented by gray, green and red rectangles, respectively. At the start of the loop (a) the Hi-1,j-1 value was already used in computations of the previous 12-element block and is remembered in a register. The Hi-1,j and Fi-1,j values are read from global memory, using one 32-bit read operation. The Ei,j-1 and Hi,j-1 values are read from shared memory also using one 32-bit read operation. The Ai,j is also read from shared memory. Once values of Fi,j, Ei,j and Hi,j are computed, the Hi-1,j is stored in a register for initialisation of next block and values of Ei,j and Hi,j are stored in the shared memory, using one 32-bit write operation. One may notice, that writing values of Ei,j and Hi,j is performed at all steps of the inner loop. Then for next ten steps (b) algorithm neither writes or reads from global memory. Each step is similar to the initalisation, but here Hi-1,j-1, Hi-1,j and Fi-1,j values are already present in the registers from previous iteration, therefore only Ai,j, Ei,j-1 and Hi,j-1 values are read from the shared memory. Finally in the last step (c) values of Hi,j and Fi,j are written into the global memory using a single 32-bit operation.

was run both in a single and dual core version. To estimate the asymptotical performance of the code we ran also two runs on two synthetic databases. The first database comprised of 81920 identical sequences, each 1000 residues long and second one equal number of random sequences of the same length. The first case is an ideal case for shared memory access, since all threads read the same location from the shared memory when accessing similarity score. The second case represents ideally aligned independent sequences.

III. Results and Discussion

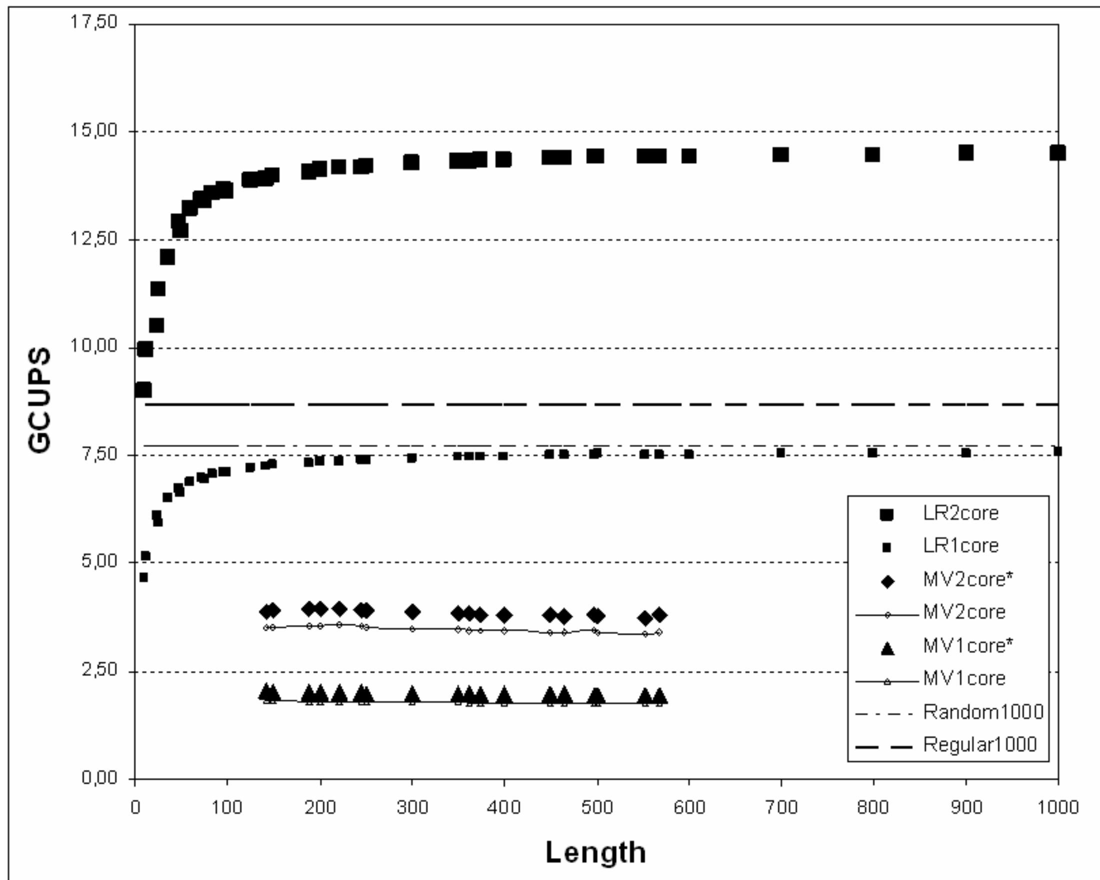
The measured performance in GCUPS is displayed in Figure 4. For a comparison we show the performance of our code, along with the reported results for earlier implementation of SW algorithm on GPU.

The GPU implementation of the SW algorithm presented here is significantly improved over the previous version. The minimal performance was observed for query sequence only 10 residues long. It was 4.65 and 8.99 GCUPS for single and dual core configuration, respectively. For more practically relevant short query lengths (more than 30 residues) performance was higher than 6 and 12 GCUPS for single and dual core configurations, respectively. The maximal performance, approaching 7.5 and 14.5 GCUPS for single and dual core configurations, respectively, was achieved for longest queries. This performance was close to the performance obtained for the random synthetic database, the latter being equal to 7.74 GCUPS. Performance measured on the ideal case synthetic database was 8.67 GCUPS. Both results were obtained for a single core.

Our implementation is computationally bound. Additional multiplication (equivalent to four additions) and one addition are required for address computation, increasing the total operation count to 16. This leads to the theoretical estimate of maximal achievable performance at 192 / 16 = 12 GCUPS. The maximal performance observed on the artificial data, where all database sequences were identical was more than 72% of that theoretical limit.

One should note that the tests were performed on the subset of database, with roughly twelve thousand longest sequences removed. This is because the current implementation is a proof of concept rather than a production quality code. The technical constraints will be removed in further development of the code. Nevertheless, with current implementation, including these sequences in a straightforward way would decrease achieved performance. This is because the number of proteins having the same length decreases quite rapidly with increasing length. This leads to a situation when many threads wait idle for the thread processing the longest sequence. The estimated decrease of performance on SwissProt database would be at around 5.5% in comparison with the results reported here. This reduction can be avoided either by removing the small fraction (slightly more than 1%) of longest sequences from the database and scanning them concurrently using CPU version of SW algorithm. Alternatively one could divide the longest sequences into overlapping segments, with length of each segment longer than that of largest known protein domains.

Our results are not directly comparable to the results of Manavski and Valle, since they used a different card. Nevertheless, since the architecture of a single core in 9800 GX2 card is very similar to that of the 8800 GTX card, a fair comparison can be taken by scaling up their results by a factor resulting from the main clock frequency difference (1.5 GHz versus 1.35 GHz). Both original and scaled up results are presented in Figure 4.



The difference of performance between the current and previous implementation arises due to different usage of memory. As discussed earlier, in our implementation the global memory is accessed only at the loop initialization and for writing the results at the exit. This is quite different from the implementation of Manavski and Valle. Their implementation uses global memory intensively. Their algorithm uses one 128-bit write and one 128-bit read operation per four cell updates. This is equivalent to 8 bytes of global memory read/write operations per cell update. The 128-bit operations are the least efficient method of accessing data [30], two times slower than theoretical limit. Taking that into account the memory bandwidth limits the theoretical performance of this implementation to at most 4 GCUPS. Moreover, the additional operations necessary for the packing and unpacking take additional registers, lowering the number of threads that can be executed concurrently on a single multiprocessor. This in turn may expose the memory access latencies, limiting effective bandwidth even more.

This difference in memory usage pattern resulted also in different organization of the code. The optimal kernel configuration in our implementation is 4096 threads in 16 blocks (256 threads per block). Thanks to the optimization of register usage we managed to use 256 threads in a single block, what allows for efficient hiding of the latency of main memory access. Nevertheless due to limited number of available registers we were unable to use full throughput of the multiprocessor, which is utilized best when the maximal number of threads is executing.

In the previous implementation the optimal configuration was 64 threads per block with 450 blocks, what gives the total number of threads equal to 28 800. One should note, that this number is higher than a maximal number of threads which can execute concurrently in hardware, which is 12 288 [31].

IV. CONCLUSIONS

Total performance of the dual core Nvidia 9800 GX2 card is 14.5 GCUPS, which is currently the highest performance of the SW algorithm on commodity hardware, roughly twice the estimated performance on quad core Pentium processors with Farrar implementation [25] and more than 50% faster than our own implementation on SonyPlaystation 3 [28]. It is also comparable to the performance of BLAST heuristic on a single core Pentium processor. One should note that CPU and PS3 implementations rely on vector instructions and use only one byte to represent numbers, whereas GPU version uses 2-byte

representation of integers (full 4-byte representation can be used with a small performance decrease). Therefore these vector implementations are suitable only for scanning large databases of non-similar sequences. Once the hit is found they have to call the exact version of the algorithm to generate alignment. On the other hand the GPU implementation uses full integer representation of numbers and can be easily extended to return alignment, with small decrease of performance.

This makes GPU implementation suitable for applications where many similar sequences should be aligned – as for example in DNA assembly problems or in multiple sequence alignment.

The results of this project are useful for bioinformatics, where one can find several practical applications of the very efficient implementation of the SW algorithm, but also have some further reaching implications. The Berkeley team called in their report [1] for development of tools, which can simplify design of massively parallel applications. The SW algorithm is relatively simple and very well known. Nevertheless, achieving a full potential of massively parallel solution requires rewriting the algorithm, taking into account several conflicting architectural constraints. The resulting codes may differ in performance by almost one order of magnitude. It is rather unlikely that achieving high performance level can be achieved by writing the algorithm in a generic way and then letting the compiler of some high-level language do the optimization, protecting programmer from working with the details of the architecture. On the other hand, once an efficient implementation has been written on the low level by a skilled programmer, it is straightforward to use such implementation in any other algorithm, like a standard library call.

Recently the specification of the OpenCL environment for performing computations on a range of multicore computing devices has been published [32]. The overall design of OpenCL is very similar to CUDA and all CUDA compliant GPUs will be capable to run OpenCL codes. Therefore porting of our code to OpenCL should be straightforward. This extends possible applications to any hardware platform supporting OpenCL, including for example GPUs from AMD or Cell processors.

REFERENCES

A. *References*

1 Asanovic, K., et al., The Landscape of Parallel Computing Research: A View from Berkeley, in Electrical Engineering and Computer Sciences, Technical Report No. UCB/EECS-2006-183. 2006, University of Calafornia at Berkeley: Berkeley.

2. Vogt, L., et al., Accelerating Resolution-of-the-Identity Second-Order Møller−Plesset Quantum Chemistry Calculations with Graphical Processing Units. Journal of Physical Chemistry A, 2008. **112**(10): p. 2049-2057

3. Yasuda, K., *Accelerating Density Functional Calculations with Graphics Processing Unit.* Journal of Chemical Theory and Computation, 2008. **4**(8): p. 1230-1236.

4. Molemaker, J., et al. Low Viscosity Flow Simulations for Animation. in Eurographics/ ACM SIGGRAPH Symposium on Computer Animation. 2008. Dublin: The Eurographics Association.

5. Tölke, J., Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. Computing and Visualization in Science, 2008. **online first**.

6. Tölke, J. and M. Krafczyk, *TeraFLOP computing on a desktop PC with GPUs for 3D CFD.* International Journal of Computational Fluid Dynamics, 2008. **22**(7): p. 443 - 456.

7. Schive, H.-Y., et al., *Graphic-card cluster for astrophysics (GraCCA) Performance tests.* New Astronomy, 2008. **13**(6): p. 418-435.

8. Lieberman, M.D., J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. in 24th IEEE International Conference on Data Engineering. 2008. Cancun, Mexico: IEEE.

9. Schatz, M., et al., High-throughput sequence alignment using Graphics Processing Units. BMC Bioinformatics, 2007. **8**(1): p. 474.

10. Manavski, S. and G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics, 2008. **9**(Suppl 2): p. S10.

11. Dayhoff, M.O., R.M. Schwartz, and B.C. Orcutt, *A model of evolutionary change in proteins.*, in *Atlas of Protein Sequence and Structure*, M.O. Dayhoff, Editor. 1978, National Biomedical Research Foundation.

12. Henikoff, S. and J.G. Henikoff, *Amino acid substitution matrices from protein blocks.* Proc Natl Acad Sci U S A, 1992. **89**: p. 10915 - 10919.

13. Altschul, S.F., et al., *Basic Local Alignment Search Tool.* Journal of Molecular Biology, 1990. **215**: p. 403-410.

14. Altschul, S.F., et al., Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res, 1997. **25**(17): p. 3389-402.

15. Smith, T.F. and M.S. Waterman, *Identification of common molecular subsequences.* J Mol Biol, 1981. **147**(1): p. 195-7.

16. Pearson, W., Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. Genomics, 1991. **11**: p. 635 - 650.

17. Pearson, W. and D. Lipman, *Improved tools for biological sequence comparison.* Proc Natl Acad Sci USA, 1988. **85**: p. 2444 - 2448.

18. Rognes, T., PARALIGN: rapid and sensitive sequence similarity searches powered by parallel computing technology. Nucleic Acids Research, 2001. **29**: p. 1647-1652.

19. Rychlewski, L., et al., Comparison of sequence profiles. Strategies for structural predictions using sequence information. Protein Science, 2000. **9**: p. 232–241.

20. Margulies, M., et al., Genome sequencing in microfabricated high-density picolitre reactors. Nature, 2005. **437**(7057): p. 376-380.

21. Mardis, E.R., The impact of next-generation sequencing technology on genetics. Trends in Genetics, 2008. **24**: p. 133-141.

22. Blazewicz, J., et al., A new algorithm for genome assembly from short reads in 1st-International-Conference-on-Information-Technology-IT-2008. 2008, IEEE: Gdańsk, Poland.

23. Wozniak, A., Using video-oriented instructions to speed up sequence comparison. Comput Appl Biosci, 1997. **13**(2): p. 145-50.

24. Rognes, T. and E. Seeberg, Six-fold speed-up of Smith-Waterman sequence database searches using parallel

processing on common microprocessors. Bioinformatics, 2000. **16**(8): p. 699-706.
25. Farrar, M., Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics, 2007. **23**(2): p. 156-61.
26. Wirawan, A., et al., *CBESW: Sequence Alignment on the Playstation 3*. BMC Bioinformatics, 2008. **9**(1): p. 377.
27. Alpern, B., L. Carter, and K.S. Gatlin. Microparallelism and High-Performance Protein Matching. in ACM/IEEE Supercomputing Conference. 1995. San Diego, California.
28. Rudnicki, W.R., et al., The new SIMD implementation of the Smith-Waterman algorithm on Cell microprocessor. Fundamenta Informaticae, in press.
29. Liu, W., et al., *Bio-Sequence Database Scanning On GPU*. Proceeding of the 20th IEEE International Parallel & Distributed Processing Symposium: 2006(IPDSP 2006) (HICOMB Workshop, 2006.
30. NVIDIA, NVIDIA CUDA Compute Unified Device Architecture, Programming Guide version 1.0. 2007: Nvidia.
31. NVIDIA, Technical Brief NVIDIA GeForce® GTX 200 GPU Architectural Overview Second-Generation Unified GPU Architecture for Visual Computing. 2008.
32. Khronos OpenCL Working Group. Munshi, A., Editor, The OpenCL Specification Version: 1.0.