Global Sequence Alignment using CUDA compatible multi-core GPU

T. R. P. Siriwardena & D. N. Ranasinghe
University of Colombo School of Computing, Sri Lanka
ranga.prasa@gmail.com, dnr@ucsc.cmb.ac.lk

***Abstract*—The GPU has become a competitive general purpose computational hardware platform in the last few years. Recent improvements in GPUs highly parallel programming capabilities such as CUDA has lead to a variety of complex applications with tremendous performance improvements. Genetic Sequence alignment is considered to be one of the application domains which require further improvements in the execution speed, because it is a computationally intensive task with increased database size. We focus on using the massively parallel architecture of GPU as a solution for the improvement of sequence alignment task. For that purpose we have implemented a CUDA based heterogeneous solution for the global sequence alignment task with Needleman-Wunsch dynamic programming algorithm. We have compared different levels of memory access patterns to identify better parallelization strategy with different ways of kernel access and thread utilization methods.**

I. Introduction

Sequence alignment of genetic databases is considered to be a very common and essential task in the field of Bio-Informatics. Similarities between sequences are mostly obtained with maximum sensitivity using optimal alignment approach. Needleman-Wunsch algorithm [1] for global alignment and Smith-Waterman algorithm [2] for local alignment are widely used dynamic programming based approaches for that purpose. But the growth of database size increase the time required for searching using this kind of dynamic programming approaches [5].

Heuristic methods such as FASTA and BLAST [3] are implemented as a solution for this issue and shows that they are up to 40 times faster than the normal CPU based implementation of Smith-Waterman algorithm for local alignment. Even though this CPU based solution such as FASTA and BLAST are somewhat efficient and faster, still they are time consuming and take considerable amount of time for sequence database searching.

Over the past few years GPU (Graphics Processing Unit) have become competitive computing hardware against the CPU (Central Processing Unit) because of its increased performance and capabilities. Recent improvements of GPU's highly parallel programming capabilities such as CUDA [8] have lead to the mapping of wide variety of complex application with tremendous performance improvements. This is called General Purpose Computation on Graphics Processor (GPGPU) [6] and this feature leads GPU to the next generation of high performance computing.

However GPU programmers still having lot of barriers, when using GPU as a general purpose computational processor because these chips are specially designed for game development. Still the GPU programming models are unusual, the programming environment is tightly constrained and the underlying architectures are largely secret. Because of these barriers porting code written for the CPU to the GPU is not a straightforward task. But with the introduction of Compute Unified Device Architecture (CUDA) there emerged a great solution for this problem and it provides an extension of the C language with high level interface, to the programmer.

In recent years there were number of attempts to improve performance of local alignment using Smith-Waterman algorithm with CUDA [4]. These research evidently proved that CUDA compatible GPU cards are now advanced enough to be considered as efficient hardware based accelerators for sequence alignment task using dynamic programming approaches with greatest sensitivity and are allowing normal PCs to do alignment tasks. Also these implementations outperform the CPU based local alignment solutions such as BLAST.

Even though global sequence alignment is considered to be the most resource consuming and computationally intensive task than local alignment, there are fewer research works on the global alignment task with CUDA. "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA" [11], demonstrates a dynamic programming based sequence alignment approach with the use of CUDA, where they have mentioned that their implementation is up to 2.9 times faster than the single threaded CPU based implementation.

The main focus of this research work is to evaluate the efficiency of CUDA compatible GPU cards for the implementation of global sequence alignment with Needleman-Wunsch algorithm with different levels of memory access strategies. Firstly we evaluate the global memory based non-blocking strategy and then we evaluate the shared memory and global memory

T. R. P. Siriwardena is a student of the University Of Colombo School Of Computing and following the Bachelor degree in Computer Science.

combined blocking strategy. With these strategies we identify an efficient way of use, per-thread local memory, per-block shared memory and the global memory.

CUDA performance is compared against its CPU based serial version with varying lengths of sequences. We also compare GPU based implementations against each other and have identified better memory access strategies among them.

Rest of this paper is organized as follows. First few section of the paper describes Needleman-wunsch algorithm and the design of the parallel algorithm. Then we describe our implementation, experimental setup, the set of evaluation results and the conclusion.

II. Needleman-Wunsch (NW) Algorithm

Sequence similarity searching between biological macromolecules is considered to be a ubiquitous good in molecular biology. Today comparative sequence analysis is highly used to understand genomes, RNA's and proteins. In practice finding the optimal alignment between two sequences can be a computationally demanding task. Dynamic Programming (DP) provides possible solutions for this. DP makes sequence alignment task tractable as long as we follow few rules [9].

NW algorithm [1] is considered to be one of the widely used global sequence alignment algorithms based on dynamic programming. In this algorithm, alignment takes place in a two-dimensional matrix in which each cell relates to the pairing of one letter from each sequence. Each cell of the matrix holds two values: a score and pointer. Score is derived from a scoring schema and the pointer is a directional navigator that points left, up or diagonal. The alignment begins from the upper left and follows a mostly diagonal path down and to the right. An important feature related to the global alignment is every letter of each sequence is aligned to a letter or a gap. NW algorithm consists of three steps as below.

1) Initialization: The first row and column is initialized with a score. The score is set to the gap score multiplied by the distance from the origin. Pointer of each cell is point back to the origin because It is considered to be a requirement for the global alignment which guarantees that alignments go all the way back to the origin.

2) Fill: All cells in the matrix are filled with a scores and a pointer. To find the score of a cell H(i,j) we find the maximum value among three scores: a match score, a vertical gap score and a horizontal gap score Fig. reffig:fill1. The match score is the sum of the diagonal cell score and the score for a match. The horizontal gap score is the sum of the cell to the left and the gap score . Also the vertical gap score is the sum of the cell to the up and the gap score.

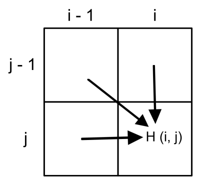

Fig. 1. Finding the score of H(i,j)

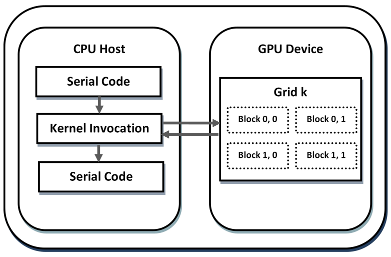

Fig. 2. Heterogeneous computing model with GPU

3) Trace-Back: Simply recovers the alignment from the matrix. Recovery process starts from the bottom-right corner and follow the pointer until get in to the beginning. In this case we are following from end to start and because of that alignment will be backward.

III. Algorithms

GPU computing allows general purpose computing on GPU together with CPU. This heterogeneous computing model allows executing the serial part of the program on the CPU and computationally intensive part on the GPU as shown in fig. 2.

When considering about each step of the algorithm the fill step is the most computationally intensive part and because of this reason parallel execution of fill step will cause to speed-up the whole algorithm. Our main focus is to explore parallelism with fill step and allow it to execute inside GPU. Initialization and Trace-Back steps will execute as a serial fraction without any involvement of GPU.

A. Parallelism inside the NW algorithm

As described in the Fill step of the NW algorithm when calculating the score of a cell it needs to know the values on its left, upper and upper-left cells [4] as depicted in the fig. 3. This dependency makes it harder for parallel execution of the algorithm. But the Fill step of the algorithm shows a pattern for parallel execution as shown in the fig. 4. According to that pattern, scores of all cells on each minor-diagonal are independent from each other. Because of this reason, all cells on each minor-diagonal can be computed in parallel [4].

B. Fill Step with blocking strategy

In order to achieve better performance from CUDA based NW alignment algorithm it should be important to have an efficient memory access pattern. So we

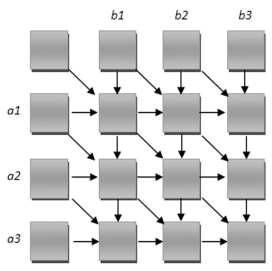

Fig. 3. Data dependency of NW algorithm

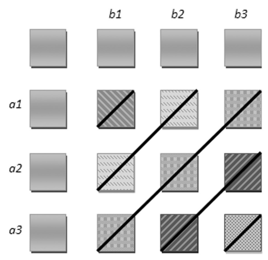

Fig. 4. Pattern of parallelism of NW algorithm

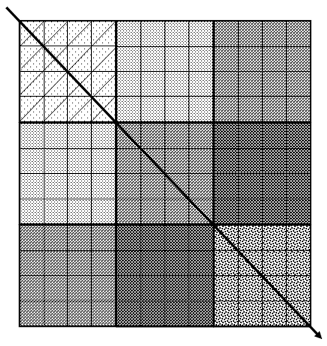

Fig. 5. Fill step with blocking strategy

have to concentrate on efficient use of per-thread local memory, per-block shared memory, constant memory and the global memory. And also the amount of communication between main memory and device memory should be minimized.

When considering about efficient memory access patterns blocking can be considered as a good strategy with CUDA. One of the solutions based on blocking is to use a minor-diagonal-wise blocking strategy [11] as shown in the fig 5. Two levels of of parallelism can be identified with this strategy as bellow.

- parallelism with threads within a single block
- parallelism among several blocks

IV. Implementation

A. NW parallelization with global Memory access

Firstly we concentrated on NW based parallel implementation which uses the global memory of GPU. Initialization and Trace-Back steps execute as CPU code and the Fill step is computed after invoking the GPU. Here we used the parallelism with minor-diagonals. With this implementation the CPU is responsible for the minor diagonal management and GPU only concentrates on the calculating the assigned cells.

Here the access to minor diagonals is managed by the host (CPU). Also copying the required data from host to device and device to host is managed by the host. In addition to score matrixes two sequences and their lengths are needed to copy from host to device before invoking the kernel.

With the host based minor-diagonal management the CUDA kernel doed not need to have extra thread synchronizations. The device only needs to calculate assigned minor diagonal in parallel way. With our kernel implementation, to fill a cell, all memory read and write operations are dealt with the global memory.

B. NW parallelization with shared Memory access

In the next step we concentrated on NW based parallel implementation which is based on the blocking strategy. With this we use both global and shared memory of the GPU and main intention is to improve the global memory based implementation with efficient use of on chip shared memory. Here we select blocks in each minor diagonal for parallel computation.

Then the device calculates its assigned blocks in parallel. With our implementation firstly, threads initialize the shared memory block using the values in global memory. Then threads are involved to calculate block in minor-diagonal-wise cell pattern inside the block. Here it uses thread synchronization explicitly to guarantee and order calculation of minor-diagonals. After the block is filled it copies the result from shared memory to global memory in parallel. Throughout the implementation thread synchronization barriers should carefully used to avoid data conflicts inside the CUDA kernel.

V. Experimental setup and hardware configuration

All the experiments were conducted on a PC with a 2.4 GHz Intel Quad Core processor, 3GB RAM and running Linux OS. A single Nvidia GeForce 8800 GT GPU was plugged to this PC and this GPU has 512MB graphics memory. GPU consists of 114 cores and 16KB of shared memory per block. Nvidia Graphics driver version 2.3 was installed for getting CUDA compatibility.

VI. Evaluation and Discussion

In this section we discuss the evaluation results and some discussion based on those results.

A. Results

1) Global memory based implementation : Here we evaluated the original NW implementation which is based on global memory access. We compared the execution time for the computationally intensive fill

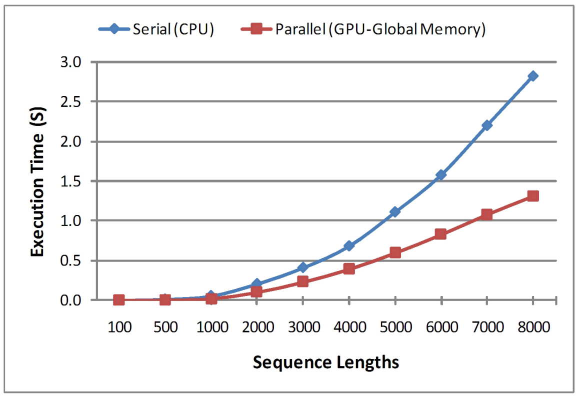

Fig. 6. Execution time comparison of CPU based implementation and GPU-Global memory based implementation

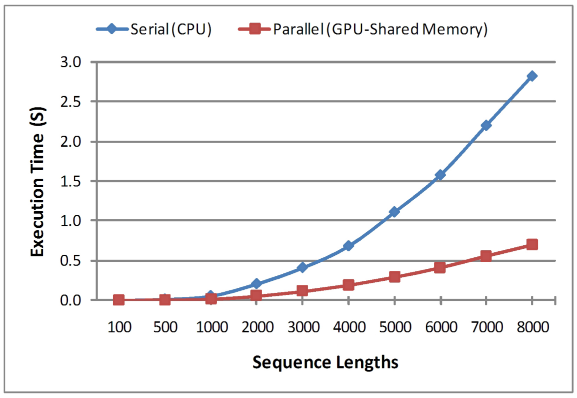

Fig. 7. Execution time comparison of CPU based implementation and GPU-Shared memory based implementation

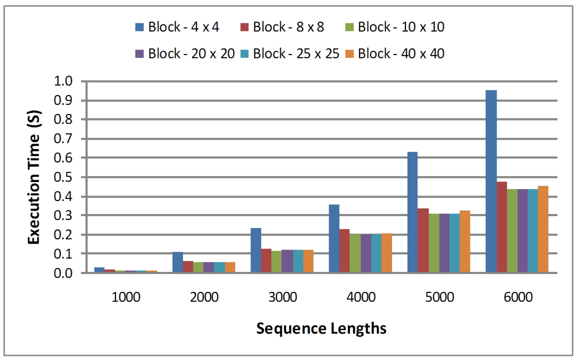

Fig. 8. Execution time comparison of GPU-Shared memory based implementation with various shared memory block size

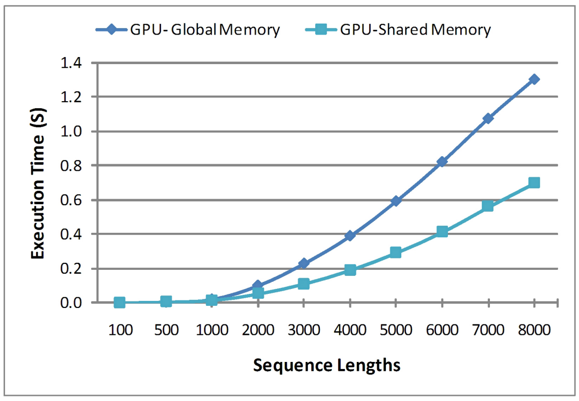

Fig. 9. Execution time comparison of GPU-Global memory based implementation and GPU-Shared memory based implementation

step of the algorithm in two aspects. They are CPU based implementation and GPU based implementation. With this evaluation we did not compare the initialization and trace-back steps because both of the implementations use the same serial execution and they are not intensive as much as fill step. Fig. 6 shows the performance derived from those implementations.

2) Shared memory based implementation : Blocking strategy based implementation is the next level of implementation and it uses shared memory and global memory combined approach. Main concern was to use the on chip shared memory to reduce the global memory access cost as much as possible. This implementation was compared against CPU based implementation and the output of comparison is shown in fig. 7.

Then we measured the performance of this shared memory based implementation with various block sizes and various sequence lengths. Fig. 8 shows the output for various block size of shared memory allocation.

3) Global memory vs Shared memory: After the comparison with CPU based implementation both of the global and shared memory based implementations were compared. Fig. 9 shows the execution time for both implementations.

B. Discussion

Even though CUDA allows us to execute programs by launching hundreds of light weight threads, the amount of performance improvement we can obtain depends on the design of parallel algorithm. Efficient use of CUDA memory levels requires rethink of the algorithm. Only using device memory which is implemented with dynamic random access memory (DRAM), without efficient use of other memory levels may lead to poor performance. The reason for this poor performance is the device memory's long access latencies and finite access bandwidth.

According to the fig. 6 it is very clear that even with the use with device memory the algorithm achieves better performance than the CPU based serial implementation. We compared the performance of two counter parts for varying lengths of sequences and identified that for longer sequences, CUDA based implementation provides better performance in its execution.

From the shared memory based implementation we further tried to improve the performance of algorithm with efficient use of shared memory. But that's where we have to focus on memory limitations of CUDA memory architecture. With CUDA, global memory is large but slow and shared memory is fast but

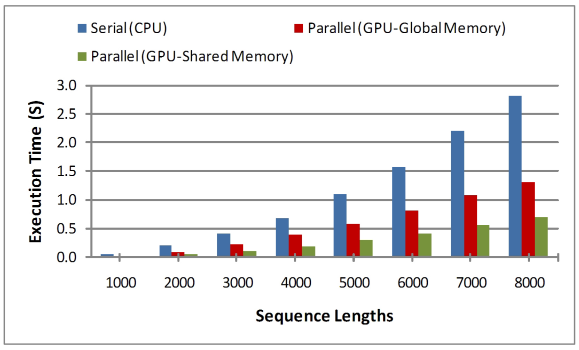

Fig. 10. Execution time comparison of CPU based implementation, GPU-Global memory based implementation and GPU-Shared memory based implementation

small. To deal with this tradeoff data partitioning as blocks was used and calculation of fill matrix as performed in block wise pattern. Fig. 7 shows the further performance improvement achieved for the NW algorithm.

Fig. 9 depicts the difference between device memory based and shared memory based implementations. It is very clear that with the use of faster shared memory the performance of algorithm further improves and fig. 10 shows the summary of performance comparisons between all implementations.

CUDA provides a limited amount of memory and programmer must be careful not to exceed those memory limits. These memory capacities are application dependent. And also CUDA limits the number of registers which can be use by a single thread. Over use of this registers may lead to reduction of amount of concurrent threads within a streaming multiprocessor(SM).

With our implementation we have identified that the largest shared memory block size we can obtain is 44 x 44. When we exceed this amount, nvcc compiler notifies about the over usage of shared resources. We measured the fill step execution time for various size of shared memory blocks and its result is shown in fig. 8. According to the result it is very clear that, performance of the algorithm increases with larger block sizes. And also the performance improvement from the block size of 8 x 8 do not show a big difference so the amount of computation is good enough to avoid the performance hit of extra computations and communication between shared memory and global memory.

Final observation regarding the analysis is to identify the capability of using GPU as a platform for the global sequence alignment task. When looking at the obtained results, there is no doubt that GPU offers a better solution for the global sequence alignment task even though it does not produce superior speed up like general matrix multiplication problem. We can conclude that even with data dependent algorithms, GPU is offers better performance with proper parallel algorithm design.

VII. Conclusion

In this research we used the massively parallel architecture of GPU as a solution for the global sequence alignment with the support of CUDA. Though the Dynamic programming based NW algorithm has low data parallelism the parallelism inside the minor-diagonal wise strips can be used with non-blocking and blocking strategies. For the non-blocking strategy we achieves up to 2 times speed up and with blocking we achieves up to 4.2 times speed up compared to the CPU based implementation. For data dependant applications there are ways to achieve better performance with different approaches in parallel application design with use of massively parallel architecture. Even though GPU based global sequence alignment algorithm implementations do not yield superior performance as much as data independent applications, they are still good enough to outperform the processing capabilities of CPU based implementation.

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," Journal of molecular biology, vol. 48, no. 3, pp. 443-453, March 1970. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/5420325

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," Journal of Molecular Biology, vol. 147, pp. 195-197, 1981. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.7142

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," Journal of molecular biology, vol. 215, no. 3, pp. 403-410, October 1990.

[4] S. A. Manavski and G. Ville, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," BMC Bioinformatics, March 2008.

[5] W. Liu, B. Schmidt, G. Voss, A. Schroder and W. Muller-Wittig, "Bio-Sequence database scanning on a GPU," Nanyanga Technological University, Singapore, 2006.

[6] J. D. Owens, M. Huston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU computing," University of California, Davis, May 2008.

[7] Y. Liu, D. Maskell, and B. Schmidt, "Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units," BMC Research Notes, vol. 2, no. 1, pp. 73+, 2009. [Online]. Available: http://dx.doi.org/10.1186/1756-0500-2-73

[8] NVIDIA CUDA Programming Guide Version 2.3.1, 2009.

[9] I. Korf, M. Yandell, and J. Bedell, Blast. O'Reilly Media, Inc., June 2003. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0596002998

[10] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 1st ed. Morgan Kaufmann, February 2010. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0123814723

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.4849