Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability

Amine Dhraief
HANA Research Group
University of Manouba, Tunisia
Email: amine.dhraief@hanalab.org

Raik Issaoui
HANA Research Group
University of Manouba, Tunisia
Email: raik.aissaoui@gmail.com

Abdelfettah Belghith
HANA Research Group
University of Manouba, Tunisia
Email: abdelfattah.belghith@ensi.rnu.tn

***Abstract*—Sequence alignment is one of the most used tools in bioinformatic to find the resemblance among many sequences like ADN, ARN, amino acids. The longest common subsequence (LCS) of biological sequences is an essential and effective technique in sequence alignment. For solving the LCS problem, we resort to dynamic programming approach. Due to the growth of databases sizes of biological sequences, parallel algorithms are the best solution to solve these large size problems. Meantime, the GPU has become an important element for applications that can benefit from parallel computing. In this paper, we first study and compare some languages for parallel development on GPU (CUDA and OpenCL). Then, we present a parallelization approach for solving the LCS problem on GPU. Finally, we evaluate our proposed algorithm on an platform using CUDA, OpenCL and on CPU using the C Language and the OpenMP API. The experiment results show that the implementation of our algorithms in CUDA outperforms the implementation in OpenCL, and the execution time is about 17 times faster on GPUs than on typical CPUs.**

***Index Terms*—Bioinformatic, GPU, Parallel algorithm, LCS, CUDA, OpenCL.**

I. Introduction

Most common studies in the bioinformatic field have evolved towards a more large scale, passing from the analysis of a single gene/protein to the study of a genome/proteome, from a single mechanism within the body towards the biology of the entire system. Hence, it become more and more difficult to achieve these analyses on a single computer, and even for some of them on clusters. The bioinformatic requires now more infrastructure allowing the storage, the transfer of huge amount of data, and finally, the massive computation for their analysis.

Until recently, the CPU, or the computer main chip, dealt most heavy load operations such as physics simulation, bioinformatic, rendering off-line for movies, calculations of risks for financial institutions, weather forecasting, file encoding video and audio [13]. Some of these heavy computations [10] however, are easily parallelizable and can therefore benefit from an architecture for parallel computing. Most parallel architectures were heavy, expensive and targeted at a specific market.

That is until the Graphic Processing Unit (GPU) imposes itself as a major player in the parallel computing [11]. The graphics processors have rapidly evolved, with continual growth performance, more generic architectures and high-level programming tools (CUDA, OpenCL, etc.). GPUs are processors that can run a smaller job a great many times in parallel, while the CPUs are expected to perform lengthy and complex calculations in series [4]. The General Purpose Computation on Graphic Processing Unit (GPGPU) [9] is a new low-cost technology which take advantage of the GPUs to perform massively parallel calculation, traditionally executed on multi-cores CPU.

The GPU computing is designed to accelerate massively parallel algorithms taking advantage of the many execution units present in a GPU. Streaming algorithms are massively parallel algorithms, widely investigated on GPUs. They are algorithms of treatment of data stream in which the input is presented as a sequence of object and can be processed in some passes (sometimes only one). The streaming algorithms are characterized by strongly parallel computations with a little of re-use of input data. These algorithms must be designed to be decomposed in a multitude of small threads that will run in parallel, in groups, on the GPU.

We are talking about thousands of threads, in contrast to the few threads in a CPU. From the software perspective, GPUs are used with more and more tools for developing scientific computing codes using high level programming languages. In front of first spectacular results in terms of reducing the computation time, the major manufacturers now offer professional solutions, dedicated to scientific calculations. Since GPU performance grows faster than CPU performance, the use of GPUs for bioinformatics is therefore a perfect match.

Sequence alignment is a fundamental technique for biologists to investigate the similarity between different species. In computational method, biological sequences are represented as strings and finding the longest common subsequence (LCS) is a widely used method for sequence alignment. Dynamic programming is a classical approach for solving LCS problem, in which a score matrix is filled through a scoring mechanism. The best score is the length of LCS and the subsequence can be found by tracing back the table. The LCS algorithms can be considered as streaming algorithms and thus can be solved using the GPU computing paradigm. In this paper, we investigate to what extend we can solve the LCS problem on GPUs using different hight level programming tools. We show that solving the LCS problem on GPUs has a speed up 17 time faster than solving the LCS problem using traditional parallel API.

The rest of the paper is organized as follows. Section 2 gives an overview and compares the two most used tools in GPU computing, CUDA and OpenCL. Section 3 defines the Longest Common Subsequence problem and presents the used parallelization approach for solving this problem on GPUs. Section 4 shows the experimental results for solving the LCS problem on GPUs and discusses what is the most appropriate developmental environment to solve this problem on a particular GPU device. Section 6 concludes the paper.

II. GPU Computing

Current personal computers can be viewed as multi-processor stations with shared memory. Thus, developers can easily write parallel programs on these workstation by using specific programming API. OpenMP [6] is one of the most promising parallel programming APIs. It uses a multi-threading parallelization method where a single task is divided between several threads that are executed concurrently.

Moreover, most of the current personal computers holds graphic cards which embed several graphical processors. Traditionally, these processors are exclusively used for graphics manipulation purposes. Nonetheless, with the forthcoming of new programming environments, these processors can execute highly parallel programs and intensive calculus. They are consequently called Graphic Processing Units (GPUs) and becoming serious rival to the traditional CPUs.

In the following, we present the two most used used GPUs programming environments, CUDA and OpenCL.

A. CUDA, NVIDIA

The Compute Unified Device Architecture (CUDA) environment developed by Nvidia is a high-performance vector computing environment [3].

On a typical CUDA program, data are first sent from the main memory to the GPU memory. Then, the CPU sends commands to the GPU which performs the computation kernels by scheduling the work on the available hardware. Finally Compute Work Distribution (in GPU) copies the results from the GPU memory to the CPU one via the Host Interface [9].

The hardware architecture used by CUDA consists of a host processor, a host memory and an Nvidia graphics card that supports CUDA. CUDA enabled GPUs are based on the Tesla architecture [9]. These GPUs can run in parallel several thousands of instances (threads) of a unique code (SPMD model). Threads are grouped into blocks which size is defined by the programmer. All threads within a block are executed on a Streaming Multiprocessor (SM) and communicate with each other using a shared memory. Threads in different blocks can not communicate which make the scheduling of the different blocks rapid (independent of the number of SMs used for program execution). In CUDA, a grid is a group of (thread) blocks, with no synchronization between them [8].

The programming language is based on the C language with extensions to indicate whether a portion of the code is executed on the CPU or the GPU. In CUDA, a kernel is a code (usually a function) that can be executed in the the GPU. The process of creating an executable CUDA includes 3 stages associated with the CUDA compiler nvcc. First, the program is divided into CPU section and GPU one's following pragmas inserted by the programmer. The CPU part is compiled by the compiler of the host, while the GPU part is compiled using a specific compiler. The resulting GPU code is a binary CUDA (file Cubin). Both CPU and GPU programs are then linked with libraries that contain CUDA functions for loading the binary code Cubin and send to the GPU for execution.

CUDA has a hierarchy of several types of memory: global, shared, local, texture/constant memory. The global memories are visible to all threads in all blocks, the biggest and the slowest. The shared memories are visible to all threads in a particular block, a medium size and an average speed of communication. The local memories are only visible to a particular thread, the smallest and the fastest. The texture/-constant memories are visible to all threads in all blocks and they are read-only memory. Each thread can read/write independently in registers and local memory. All threads in the same block can read/write (communicate) in shared memory. And all threads in the same grid can read/write (communicate) in global and constant memory.

B. OpenCl, Apple

OpenCL (Open Compute Language) is an open API dedicated to massively parallel computing, initially developed by Apple and the Khronos Group -a consortium of firms engaged in the development of open APIs like OpenGL.

The CUDA is a non-free proprietary software and CUDA programs can exclusively be executed on Nvidia GPUs. Apple has proposed OpenCL to exploit the power of GPUs without being locked in a range of products from a particular manufacturer (Nvidia). OpenCL can therefore be seen as an API intended to standardize the GPU computing. OpenCL is a common language to all architectures, it is not intended only to GPUs but also CPUs and covers accelerators such as the Cell (in the Playstation 3).

OpenCL basic functions are exactly the same as for CUDA: a kernel is sent to the accelerator (compute device) which is composed of ”compute units” whose ”processing elements” working on ”work items”.

Finally, CUDA compatible cards (GeForce, Quadro, Tesla) support both CUDA and OpenCL.

C. Comparison

Table I presents a qualitative comparison between CUDA and OpenCL. This comparison highlights the fact that CUDA is a more mature technology than OpenCL; whereas, OpenCL has the merit to be an open standard. While CUDA can be used only on Nvidia‘s GPU, OpenCL can be used on a wide range of GPU devices.

	CUDA	**OpenCL**
Technologies	Owner	Open
Start	2006	2008
Free SDK	Yes	Depend on vendor
Many vendors	No,only NVIDIA	Yes: Apple,AMD, IBM
Multiple OS	Yes;Windows, Linux, Mac OS X;32 and 64 bits	Depend on vendor
Heterogeneous devices	No, Only NVIDIA GPUs	Yes

TABLE I: QUALITATIVE COMPARISON BETWEEN CUDA AND OPENCL

From a memory management point of view, in both CUDA and OpenCL, the host and the device memories are separated. The device memory are hierarchical designed and must be explicitly controlled by the programmer. The memory model of OpenCL is more abstracted and supplies more way for various implementations. CUDA explicitly defines all the memories levels, whereas in OpenCL, these details are hidden as they are device dependent.

CUDA and OpenCL are similar in several aspects. They are concentrated on data parallel computing model. They provide a C-basic language customized for device programming. The device, the execution scheme and the memory models are very similar.

Most of the differences result from differences of their origin. CUDA is the technology owner of Nvidia and targets only Nvidia devices. OpenCL is open and targets several devices. CUDA has been on the market earlier than OpenCL (2006 vs. 2008) and has more support, applications, related research and products. Finally, CUDA is more documented than OpenCL.

III. The LCS problem

The extraction of the longest common subsequence of two sequences is a current problem in the domain of datamining. The extraction of theses subsequence is often used as a technique of comparison to get the similarity degree between two sequences.

A. Definition

A sequence is a finished suite of symbols taken in a finished set. If $U = \langle a_1, a_2, .., a_n \rangle$ is a sequence, where $a_1, a_2, .., a_n$ are letters, the integer n is the length of u. A sequence $V = \langle b_1, b_2, .., b_n \rangle$ is a subsequence of $U = \langle a_1, a_2, .., a_n \rangle$ if there are integers $i_1, i_2, .., i_m (1 \leq i_1 < i_2 < .. < i_m \leq n)$ where $b_k = a_{i_k}$ for $k \in [1, m]$.

For example, $V = \langle B, C, D, B \rangle$ is a subsequence of $U = \langle A, B, C, B, D, A, B \rangle$. A sequence W is a subsequence common to sequences U and V if W is a subsequence of U and of V. A common subsequence is maximal or is a longer subsequence if it is of length maximal. For example: sequences $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are the longest common subsequences of $\langle A, B, C, B, D, A, B \rangle$ and of $\langle B, D, C, A, B, A \rangle$.

B. Parallel algorithms for the LCS problem

The dynamic programming is a classical approach for solving the LCS problem. It is based on the filling of a score matrix through a scoring mechanism. The best score is the length of the LCS and the subsequence can be found by tracing back the table.

Let m and n be the lengths of two strings to be compared. We determine the length of a maximal common subsequence in $A = \langle a_1, a_2, .., a_n \rangle$ and $B = \langle b_1, b_2, .., b_m \rangle$.

We note $L(i, j)$ the length of a maximal common subsequence in $\langle a_1, a_2, ..a_i \rangle$ and $\langle b_1, b_2, .., b_j \rangle (0 \leq j \leq m, 0 \leq i \leq n)$.

$$L(i,j) \begin{cases} 0 \quad if\, i = 0\, or\, j = 0 \\ L(i-1, j-1) + 1 \quad if\, a_i = b_j \\ max(L(i, j-1), L(i-1, j)) \quad else. \end{cases} \tag{1}$$

We use the above scoring function to fill the matrix row by row (Fig. 1).

The highest calculated score in the score matrix is the length of the LCS. In Fig. 1, the length is 4. In the scoring matrix, the LCS is traced back from the highest score point (4) to the score 1.

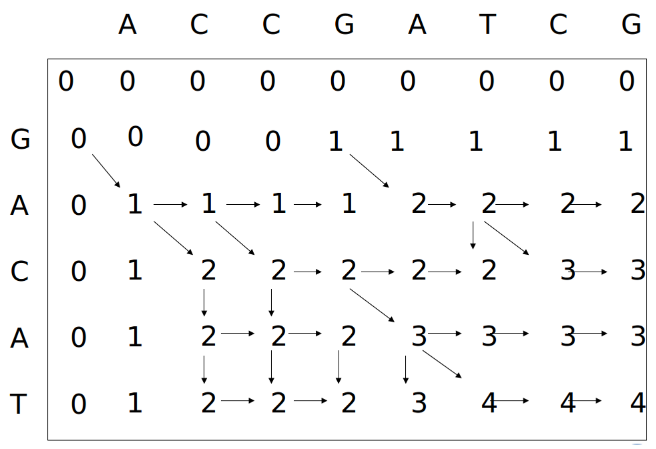

Fig. 1: Example of filling the LCS matrix score.

The time and space complexity of this dynamic programming approach is $O(mn)$, where m and n are the length of the two compared strings. Several parallel algorithms in the literature target to solve the LCS problem. In the following table, we present the parallel complexity and the number of processors of some algorithms (m and n are the length of the two compared strings, p is the number of available processor):

Reference	Parallel Complexity	Number of processors
Apotolico et al. 1990 [1]	$O(logm * logn)$	$mn/logm$
Lu and Lin 1994 [7]	$O(logm + logn)$	$mn/logm$
Babu and Saxena 1997 [2]	$O(logm)$	mn
Xu et al. 2005 [12]	$O(mn/p)$	p
Krusche et al. Tiskin 2006[5]	$O(n)$	p

C. *The parallelization approach*

In the scoring matrix that dynamic program algorithm constructs according to the equation Eq. 1, for all i and j $(1 \leq i \leq n, 1 \leq j \leq m)$, $L[i,j]$ depends on three entries; $L[i-1,j-1]$,$L[i-1,j]$ and $L[i,j-1]$ (as shown in Fig. 2). In other words, $L[i,j]$ depends on the data in the same row and the same column. So the cells in the same column or same row cannot be computed in parallel.

$$\begin{tikzcd}
L_{1,1} \arrow[r] \arrow[d] \arrow[dr] & L_{1,2} \arrow[r] \arrow[d] \arrow[dr] & L_{1,3} \arrow[r] \arrow[d] & {} \\
L_{2,1} \arrow[r] \arrow[d] \arrow[dr] & L_{2,2} \arrow[d] & {} & \\
L_{3,1} \arrow[d] & {} & & \\
{} & & {} \arrow[dr] & {} \arrow[d] \\
& & {} \arrow[r] & L_{m,n}
\end{tikzcd}$$

Fig. 2: Data dependency in the score matrix of a dynamic programming algorithm for solving LCS problem.

We start computing $L_{1,1}$ then $L_{1,2}$ and $L_{2,1}$ in the same time, afterthat $L_{1,3}$, $L_{2,2}$ and $L_{3,1}$. We continue until we fill the matrix. We notice that it is possible to compute cells in the same anti-diagonal parallelly. To parallelize the dynamic programming algorithm, we have to compute the score matrix in the anti-diagonal direction (see Fig. 3).

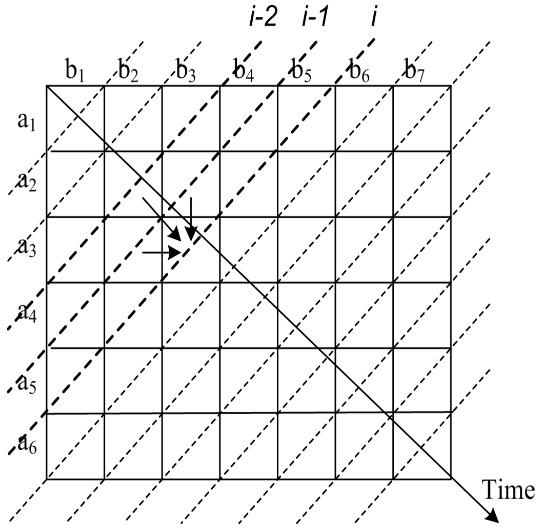

Fig. 3: The parallelization approach

IV. Results

A. *Performance measurement methodology*

We implement our LCS algorithm using CUDA, OpenCL, OpenMP and the C language on a computer equipped with a processor Intel(R) Core(TM) Quad CPU Q9400 2.66GHZ, a random access memory 8 GBytes and a graphics board NVIDIA GT 430. We use as operating system Ubuntu 10.04. The characteristics of our graphics board are listed in the following:

- GPU Engine Specifications:

CUDA Cores	96
Graphics Clock(MHz	700
Processor Clock(MHz)	1400

- Memory Specifications:

Memory C lock(MHz)	800-900 (DDR3)
Standard Memory Config	1GB DDR3
Memory Interface Width	128-bit
Memory Bandwidth(GB/sec)	25.6-28.8

We measure the filling of the LCS scoring matrix runtime without taking into account the initialization of sequences A and B. The unit of measure of the runtime is the millisecond. In all the following tests, we run as many trials as needed to reach a 95% confidence interval at $\varepsilon = 1\%$ of the average value.

B. *The execution time*

Fig. 4 illustrates the execution time of the four implementations of our LCS algorithm (CUDA, OpenCL, OpenMP and C) versus the size of the two sequences.

We can clearly see that for the C implementation of our algorithm, whenever we increase the size of the compared sequence the execution time exponentially increases. This behavior is justified by the absence of parallelization in the C implementation of our algorithm.

Both CUDA, OpenCL and OpenMP implementations of our algorithm use the parallelization approach presented in section III-C. Thus, the execution time of these three versions of our algorithm increases linearly as we increase the size of the compared sequence

For long sequences, we notice that the fastest version of our algorithm is the CUDA one, and specifically more than the OpenCL implementation. As CUDA is the SDK developed by NVIDIA for their graphic processors, it is then more appropriate for NVIDIA devices than OpenCL.

For short sequences, we plot in Fig. 5 the execution time of the four implementations of our algorithm using a logarithmic scale in the y-axis.

Fig. 5 shows that, for short sequences, OpenMP outperforms OpenCL and has similar performances than CUDA. In fact, CPU is more performant than GPU for small problems where parallelism has a negative effect on the execution time. The thread management slacken the overall execution time and signficanlty exceeds the kernel execution time.

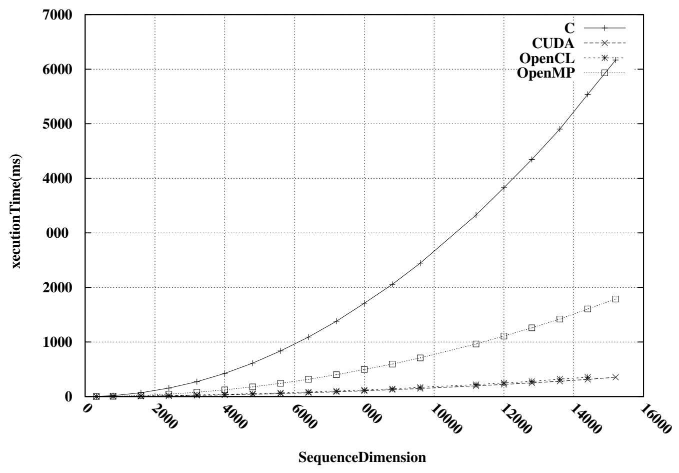

Fig. 4: The LCS execution time

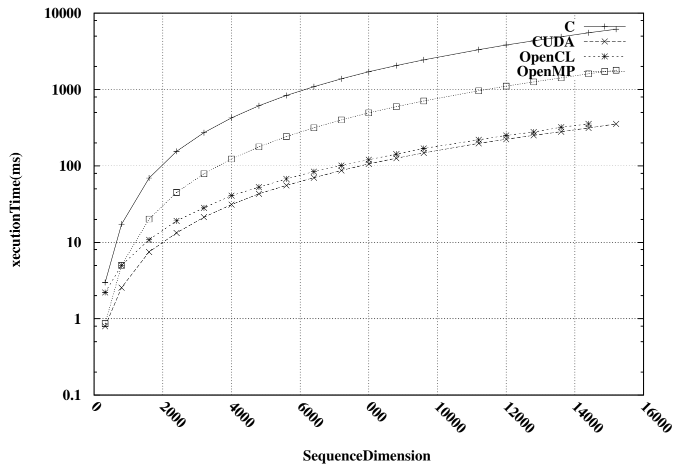

Fig. 5: The LCS execution time (logarithmic scale)

Fig. 6 compares the latency of the kernel runtime and the latency taken to transfer the data to the main memory (RAM). Only OpenCL allows us to determine these latencies.

Foremost, we notice that the required time to perform the memory transfer is the most important in the operation of

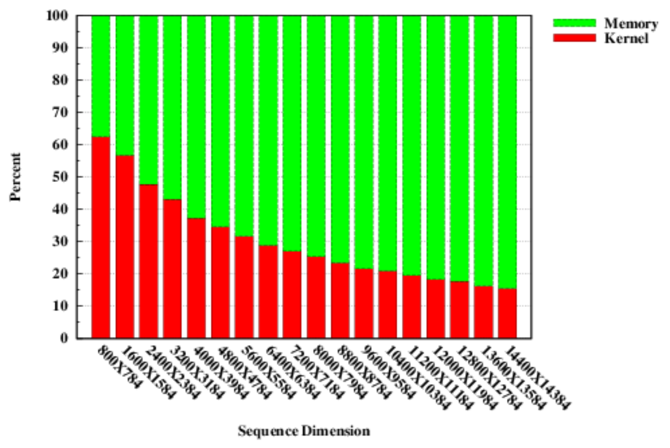

Fig. 6: Kernel latency vs. memory transfer latency

filling of the score matrix. In addition, the memory transfer latency increases with the size of the sequences.

C. The speed up

In the parallel computing, the speed up shows to what extent a parallel algorithm is faster than a corresponding sequential algorithm.

Analytically, we define the speed up as:

$$\text{SpeedUp} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}} \qquad (2)$$

Fig. 7 shows the speed up of the implementation versions of our algorithms in OpenMP, OpenCL et CUDA versus the size of the two sequences.

The measured speed up for the implementation of our algorithm in Open Mp (on CPU) is bounded at 3.22 times. In fact, we use OpenMP to get advantage of the parallelism provided by the Quad core CPU. But, because of the other operating system threads, the speed up cannot achieve 4 times

The implementation of our algorithms in CUDA outperforms the implementation in OpenCL, as we are using an NVIDIA graphic card and CUDA is specifically designed for the NVIDIA devices.

Our results show that for long sequences, the implementation of our algorithm in CUDA (on GPU) is 17 times faster than the one on CPU.

This result is explained by the CUDA architecture. In CUDA, threads are grouped in blocks (the programmer chooses their size). All threads in the same block are executed on the same Streaming Multiprocessor (SM) and communicate via a shared memory.

Our parallelism approach is based on the filling of a score matrix in the anti-diagonal sense. The threads are filling the matrix by calculating anti-diagonals one by one, every anti-diagonal uses the necessary number of block to calculate it in parallel.

We notice that up to a certain sequence size (800 characters), the GPU is under exploited. This is mainly due to the management of threads which delays the execution time. The CPU can handle with this problem effeciently for small sequences.

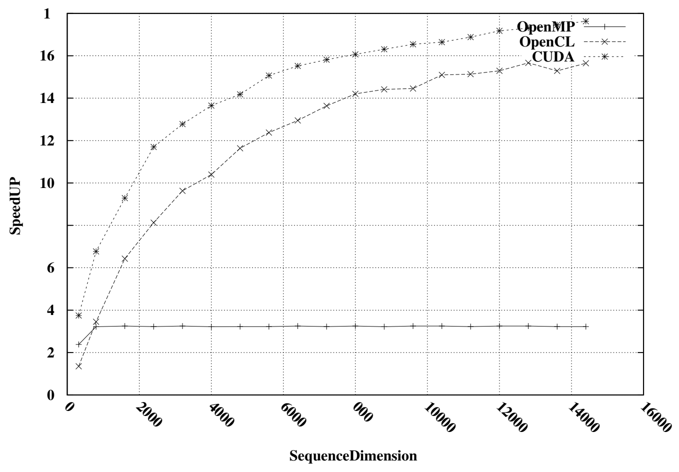

Fig. 7: Speed UP

V. Conclusion

In bioinformatic, biological sequences are represented as strings and finding the longest common subsequence (LCS) is a widely used method for sequence alignment.

We have focused in this paper on the parallelization of a dynamic programming algorithm for solving the LCS problem. For this purpose, we have studied some languages for parallel development on GPU (CUDA and OpenCL). Then, we have presented a parallelization approach for solving the LCS problem on GPU. We have evaluated our proposed algorithm on an NVIDIA platform using CUDA, OpenCL and on CPU using the C Language and the OpenMP API.

The results have shown that the studied algorithm enables higher degree of parallelism and achieves a good speedup on GPU. In addition, during this experiment, we have proved that CUDA is more suitable for NVIDIA devices than OpenCL. Finally, we have demonstrated that the major contribution in the execution time is the latency of the memory transfer from GPU Global memory to CPU RAM.

References

[1] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19:968–988, September 1990.

[2] K. Nandan Babu and Sanjeev Saxena. Parallel algorithms for the longest common subsequence problem. In *Proceedings of the Fourth International Conference on High-Performance Computing*, HIPC ’97, pages 120–, Washington, DC, USA, 1997. IEEE Computer Society.

[3] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28:13–27, July 2008.

[4] Qihang Huang, Zhiyi Huang, Paul Werstein, and Martin Purvis. GPU as a General Purpose Computing Resource. In *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, pages 151–158, 2008.

[5] Peter Krusche and Alexandre Tiskin. Efficient longest common subsequence computation using bulk-synchronous parallelism. In *ICCSA (5)*, volume 3984 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2006.

[6] Seyong Lee, Seung J. Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44:101–110, February 2009.

[7] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Trans. Parallel Distrib. Syst.*, 5:835–848, August 1994.

[8] David P. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *ISBI*, pages 836–838, 2008.

[9] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[10] James C. Phillips and John E. Stone. Probing biomolecular machines with graphics processors. *Commun. ACM*, 52:34–41, October 2009.

[11] John Shalf. The new landscape of parallel computer architecture. *Journal of Physics Conference Series*, 78(1):012066, 2007.

[12] Xiaohua Xu, Ling Chen, Yi Pan, and Ping He. Fast parallel algorithms for the longest common subsequence problem using an optical bus. In *Computational Science and Its Applications ICCSA 2005*, volume 3482 of *Lecture Notes in Computer Science*, pages 91–115. Springer Berlin / Heidelberg, 2005.

[13] Zhiguo Xu and Rajive Bagrodia. GPU-Accelerated Evaluation Platform for High Fidelity Network Modeling. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, PADS ’07, pages 131–140, Washington, DC, USA, 2007. IEEE Computer Society.