A Parallel GPU-Designed Algorithm for the Constrained Multiple Sequence Alignment Problem

Adam Gudyś and Sebastian Deorowicz

Abstract. Modern graphical processing units (GPUs) offer much more computational power than modern CPUs, so it is natural that GPUs are often used for solving many computationally-intensive problems. One of the tasks of huge importance in bioinformatics is sequence alignment. We investigate its variant introduced a few years ago in which some additional requirement on the alignment is given. As a result we propose a parallel version of Center-Star algorithm computing the constrained multiple sequence alignment at the GPU. The obtained speedup over the serial CPU relative is in range $[20, 200]$.

Keywords: constrained sequence alignment, GPU General Processing.

1 Introduction

Moore's law shows how the computational power of the top central processing units (CPUs) has been growing in the past four decades. Currently to sustain this law CPU vendors use parallelization, as the performance of a single CPU core grew only slightly in the last decade. Therefore, modern CPUs consist of 4–8 cores and this number is expected to grow in the near future.

An intensive development of graphical precessing units (GPUs) led to the situation in which the computational power of a GPU is much larger than the computational power of a CPU. An introduction of CUDA library [13] and OpenCL language [12] made this power available widely. Moreover, contemporary GPUs are designed not only for efficient processing of graphics but also (or even primarily) for easy application in general purpose computations. E.g., the 1st, 3rd, and 4th

Adam Gudyś · Sebastian Deorowicz
Institute of Informatics, Silesian University of Technology,
Akademicka 16, 44-100 Gliwice, Poland
e-mail: {adam.gudys,sebastian.deorowicz}@polsl.pl

T. Czachorski et al. (Eds.): Man-Machine Interactions 2, AISC 103, pp. 361–368.
springerlink.com

places of the Top 500 list of supercomputers from November 2010 occupy machines composed in part of NVidia's GPUs.

The algorithms designed for GPUs were proposed for various problems, some of them in the field of bioinformatics (e.g., [5, 6, 7, 8, 9, 11, 14, 15]). We investigate a problem of constrained multiple sequence alignment (CMSA), which is a multiple sequence alignment (MSA) problem with an additional requirement on the result. To the best of our knowledge there is no CMSA-solving parallel algorithm for the GPU. The CMSA is used instead of MSA when we have some prior knowledge on how the alignment should look like. The CMSA problem was formulated in [16] to compare RNase sequences.

The paper is organized as follows. Section 2 defines the CMSA problem. In Sect. 3, tools for using GPU computational power for general processing are discussed. In Sect. 4, we show Center-Star algorithm [1] which is often used to solve the CMSA problem. In Sect. 5, we introduce a parallel Center-Star algorithm for the GPU. Section 6 shows the experimental results. The last section concludes the paper.

2 Constrained Multiple Sequence Alignment Problem

Let all the sequences from the set $\mathscr{S} = \{S^1, S^2, \ldots, S^k\}$ be over a finite *alphabet* Σ. The elements of Σ are called *symbols*. The *length* of each sequence S^i is the number of symbols it contains and is denoted by n_i or $|S^i|$. Let s^i_j be the jth symbol
of S^i.

An alignment of two sequences S' and S'' is defined as a pair of equal-length sequences $\overline{S'}$ and $\overline{S''}$ such that $\overline{S'}$ and $\overline{S''}$ can be obtained from S' and S'' respectively by inserting at some positions special symbols '–' called *gaps*. Given a *distance function* $\delta(x,y)$ defined for $x,y \in \Sigma \cup \{-\}$ the *pair-wise score* of two sequences $\overline{S'}$ and $\overline{S''}$, both of length n is defined as $\sum_{1\le j\le n} \delta(\overline{s'_j}, \overline{s''_j})$. The values $\delta(x,-) = \delta(-,x) = w_g$ for any $x \in \Sigma$, where w_g is a gap cost.

A multiple sequence alignment (MSA) of $\mathscr{S}$ is a set of equal-length sequences possibly with inserted gaps: $\overline{\mathscr{S}} = \{\overline{S^1}, \overline{S^2}, \ldots, \overline{S^k}\}$. There is many more than one 'quality' measure of a multiple sequence alignment. In this paper we use the most popular *sum-of-pairs* (SP) method (see [3] for other possibilities) in which the total MSA score is a sum of sequence alignment scores for all sequence pairs: $\sum_{1\le i'<i''\le k} \sum_{1\le j\le n} \delta(\overline{s^{i'}_j}, \overline{s^{i''}_j})$, where n means the length of each of aligned sequences.

In a constrained variant of the MSA, there is an additional, *constraining*, sequence $P = p_1 p_2 \ldots p_r$. It is required that there exists an increasing integer sequence $x_1, \ldots, x_r$ ($1 \le x_1 < \cdots < x_r \le n$) such that $\forall_{1\le j\le r, 1\le i\le k} \overline{s^i_{x_j}} = p_j$. Thus, if the aligned sequences are typed one above the other, as an MSA is usually presented, the column of index x_j contains only p_j, for all valid js.

The *constrained multiple sequence alignment* (CMSA) problem is to find the constrained alignment of maximal score. For the case of $k = 2$ the problem can be solved

in $O(n_1 n_2 r)$ time [1], but in a general case of unbounded number of sequences it is NP-hard, so it can be solved exactly only for very small values of k and short sequences. Therefore, various approximate algorithms were proposed. These algorithms were often inspired by MSA-solving methods.

The first approximate algorithm was proposed by Tang et al. [16]. It is a progressive method, which can be summarized as computing a sequence alignment (SA) for each pair of sequences (without a constraining sequence) and then merging the alignments in some way to obtain the CSMA. Its worst-case time complexity is $O(rkn_*^4)$, where n_* is the longest input sequence length. Center-Star algorithm by Chin et al. [1] (see Sect. 4 for details) is much faster and gives better scores also. Its worst-case time complexity is $O(Ckn_*^2)$, where C is the total number of occurrences of the constraining sequence in all main sequences. Yet other algorithm was presented in [10]. There were also approaches on exact computation of the CMSA [1, 4] but they are rather useless for large data due to the worst-case time complexity $O(2^k rn_*^k)$.

3 General Processing at GPU

CPUs and GPUs are processors of rather different architecture. CPUs are composed of a few identical and independent cores. Each core has some levels of cache memory units. The total cache size in a CPU is in order of megabytes, while the global memory is shared by all cores and is of size of a few gigabytes.

GPUs are composed of multiprocessors.[1] Each multiprocessor contains from a few to a few tens of cores. The cores within a multiprocessor work in a SIMD-like manner, i.e., each core processes the same instruction on different data. The global memory is shared by all cores but access to it may be very slow (as is often uncached). Moreover, the accesses to the global memory should be made according to some scenario to maximize performance. There is also a small local memory separate for each multiprocessor. It is fast and cached. Finally, the number of registers per multiprocessor is large (e.g., $32,768$ in NVidia GTX 400 series).

The basic execution unit is called a thread. The threads are gathered in *blocks* (only the threads from the same block can cooperate). Each block works on a single multiprocessor but a few blocks can share the multiprocessor. A single execution of a *kernel* (a program for a GPU) consists of many blocks, and a thread scheduler allocates the blocks to the multiprocessors. The programmer should only prepare the blocks and pass them to the scheduler. The total number of threads running at a time should be at least as large as the number of cores but to maximize the performance it should be a few times larger (which means even 10^4 threads).

The GPU programming is significantly different than CPU programming (even for multicore architecture). Therefore, the algorithms for GPUs are usually designed from scratch rather than are adaptations of serial relatives.

[1] There are some differences between the GPUs designed by two dominating vendors: NVidia and AMD but the big picture is the same.

Nowadays, there are several alternatives to program the GPUs for general purposes. The most mature is NVidia's CUDA C/C++ library.[2] The main drawback of CUDA is its low portability. In 2010, the first specification of the OpenCL language was presented and then implemented by GPU vendors. The OpenCL is a language for computations in heterogeneous environment which becomes a popular solution.

4 Serial Center-Star Algorithm

The popular Center-Star algorithm [3] for the MSA problem has known approximation ratio $2(z-1)/z < 2$, i.e., the obtained score of the MSA given by this algorithm is guaranteed to be at most 2 times worse than the score of the optimal (unknown) MSA. The same holds for Center-Star algorithm by Chin et al. for the CMSA problem [1], which works as follows:

1. For a *candidate* sequence $S' \in \mathscr{S}$ find all r-tuples $\langle x_1, x_2, \ldots, x_r \rangle$, where $1 \le x_1 < \cdots < x_r \le |S'|$, such that $s'_{x_1} s'_{x_2} \ldots s'_{x_r} = p_1 p_2 \ldots p_r$. Then, compute a tuple-score for each r-tuple as a sum of constrained alignments scores between S' and all other sequence $S'' \in \mathscr{S}$. The rule for score calculation is:

$$D(i,j,\gamma) = \max \begin{cases} D(i-1,j-1,\gamma-1) + \delta(s'_i, s''_j), \text{if } x_\gamma = i, s'_i = s''_j = p_\gamma, \\ D(i-1,j-1,\gamma) + \delta(s'_i, s''_j), \\ D(i-1,j,\gamma) + \delta(s'_i, -), \\ D(i,j-1,\gamma) + \delta(-, s''_j), \end{cases} \tag{1}$$

   for $0 < i \le |S'|$, $0 < j \le |S''|$, $0 \le \gamma \le r$. The boundary conditions are: $D(i,0,\gamma) = D(0,j,\gamma) = -\infty$, $D(0,j,0) = j \times w_g$, $D(i,0,0) = i \times w_g$ for $0 < \gamma \le r$, $0 \le i < |S'|$, $0 \le j < |S''|$, where w_g is a gap cost. The best r-tuple is the one with maximum tuple-score.
2. Repeat step 1 for each $S' \in \mathscr{S}$ and find a sequence $S^* \in \mathscr{S}$ of maximal tuple-score and the related r-tuple. This is the *center* sequence.
3. Merge the other sequences with the center sequence (see [1] for details).

The most time-consuming part of Center-Star algorithm is finding the center sequence (steps 1–2). In the following subsection, we show how this process can be parallelized.

5 Our Parallel Algorithm

At the beginning of our parallel Center-Star algorithm, all valid r-tuples for each sequence of $\mathscr{S}$ are determined. This is made at CPU, since this step is very quick. Then, the descriptions of all possible tasks, each specified by a candidate sequence, r-tuple and other sequence, are stored in an array and passed to the part of the algorithm executed at GPU.

[2] See `www.nvidia.com` for a catalog of more than a thousand applications of CUDA.

Below we show how the score for a single task is determined at GPU. The n_{thr} threads compute the constrained sequence alignment (CPSA) score of two sequences with assumed positions of constraining symbols in one of these sequences (specified by r-tuple). To compute the score a 3-dimensional dynamic programming matrix according to (1) needs to be computed. It is, however, easy to notice that due to the assumed positions of the constraining symbols the space of the dynamic programming matrix are effectively reduced and for each pair of coordinates (i, j) the value of $D(i, j, \gamma)$ for exactly one γ must be computed, so we can conceptually treat D as a 2-dimensional matrix. Moreover, because we are interested only in the score, not the complete alignment, the necessary space for computations is $O(|S'| + |S''|)$.

The 2-dimensional matrix is split into strips of sizes $|S'| \times |S''|/n_{\text{thr}}$ and each thread is responsible for computation of a single strip. Due to the dependences in the dynamic programming matrix the strips must be computed in an anti-diagonal manner, i.e., if $D(i, j, \gamma)$ and $D(i, j+1, \gamma)$ belongs to different threads the thread computing $D(i, j+1, \gamma)$ must wait until $D(i, j, \gamma)$ is known. Figure 1 shows an example.

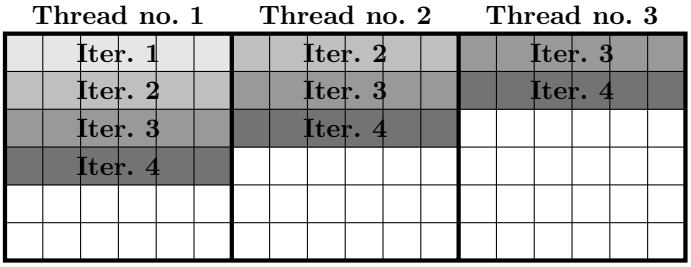

Fig. 1 Illustration of the order of computations made by threads computing a dynamic programming matrix (only the start of threads is shown). Note that consecutive threads start computations with a delay.

One of the limitations on the number of threads per a single block is a size of fast local memory. Our algorithm uses blocks of maximal possible size. This size depends on the number of threads designated to work on a single task and the length of the sequences. A block of threads usually solves more than one task. The value n_{thr} is a parameter of the algorithm, so we can even set $n_{\text{thr}} = 1$.

6 Experimental Results

The experiments were performed on a computer equipped with Intel Q6600 CPU clocked at 2.4 GHz and NVidia GTX 480 clocked at 1.4 GHz with 1.5 GB of global memory and 480 cores. The implementation was prepared in OpenCL language. We performed two sets of experiments. In the first, we used the randomly

generated main sequences of lengths: $100, 150, \ldots, 350$, where the numbers of sequences were $4, 5, \ldots, 10$, and the constraining sequence lengths were 2, 3, 4, 5. Symbols from Σ were uniformly distributed along sequences but the main sequences were guaranteed to contain a constraint. The obtained results are presented at Fig. 2. The parallel algorithm was run for various number of threads per single task and this number appended to GPU- string in figures shows the actual value of n_{thr}.

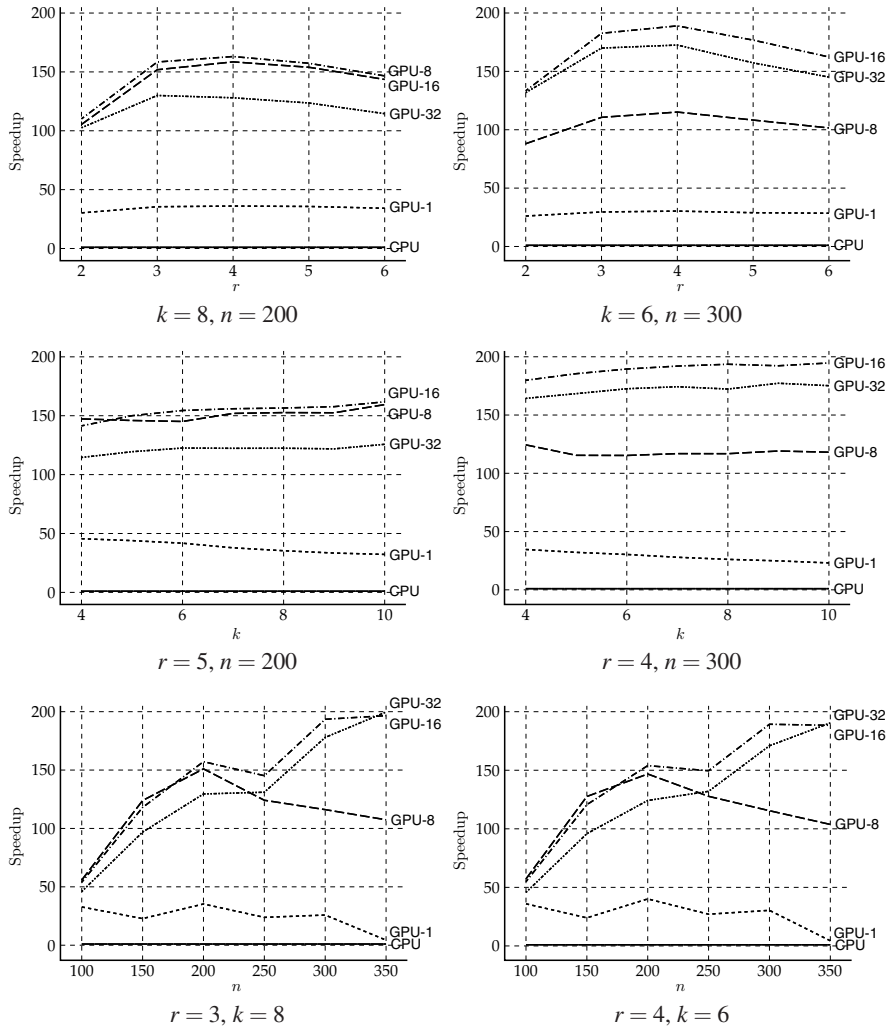

Fig. 2 Experimental results for the artificial sequences.

The results show how many times our parallel algorithm for GPU is faster than the serial CPU algorithm. The observed speedups are in the range from 20 to 200. In general, the larger the data the better the speedup. It is caused by two facts. Firstly, the larger number of sequences means much more tasks to compute at GPU and the computational power of GPU can be better utilized. Secondly, the larger the main sequences the lesser is the relative waste caused by the delayed startup of threads in the anti-diagonal computations of a dynamic programming matrix. We can also note that the speedup for $n_{\text{thr}} = 1$ was rather moderate. In this special case, the number of threads that can be allocated per each block is small, often smaller than the recommended minimal value and the computational power of multiprocessors is partially wasted. The best results are usually obtained for the case of $n_{\text{thr}} = 16$.

In the second experiment we used the real data of RNase sequences, the same as used in [1]. For the characteristics of the data please consult [2]. The results are presented in Table 1. The observations are similar here. The best is to take 16 threads for computation of each task and the speedups for this case are from 36 to 109.

Table 1 Experimental results for the real-data set. Times are in ms. Speedups (typed in bold) are calculated according to serial algorithm for CPU.

Serial (CPU) time	Parallel (GPU)							
	1 thr. per set		8 thr. per set		16 thr. per set		32 thr. per set	
	time	speedup	time	speedup	time	speedup	time	speedup
data set ds0, $P =$ HKH (7 sequences, 732 tasks)								
157.15	8.93	**17.6**	4.46	**35.3**	4.12	**38.1**	4.33	**36.3**
data set ds1, $P =$ HKH (6 sequences, 1850 tasks)								
695.68	31.63	**22.0**	8.80	**79.1**	16.09	**43.2**	14.23	**48.9**
data set ds2, $P =$ HKSH (6 sequences, 2185 tasks)								
625.62	20.91	**29.9**	11.31	**55.3**	11.01	**56.8**	11.62	**53.8**
data set ds3, $P =$ HKH (5 sequences, 3188 tasks)								
3153.82	365.38	**8.63**	41.53	**75.9**	28.92	**109.0**	28.56	**110.4**
data set ds4, $P =$ DGGG (7 sequences, 8034 tasks)								
1280.12	59.45	**21.5**	33.04	**38.7**	34.83	**36.8**	39.09	**32.7**

7 Conclusions

We presented the algorithm that computes in parallel at GPU the first and second (most time-consuming) stages of Center-Start algorithm for the constrained multiple sequence alignment problem. The experimental results show that in practice our algorithm is tens times faster (even up to 200 times) than the serial algorithm for a CPU. The results look very promising and we plan to parallelize also the last step of Center-Star algorithm to obtain a fully parallel Center-Star algorithm for the GPU.

Acknowledgements. This work was partially supported by the European Community from the European Social Fund.

References

1. Chin, F., Ho, N., Lam, T., Wong, P.: Efficient constrained multiple sequence alignment with performance guarantee. Journal of Bioinformatics and Computational Biology 3(1), 1–18 (2005)
2. Deorowicz, S., Obstój, J.: Constrained longest common subsequence computing algorithms in practice. Computing and Informatics 29(3), 427–445 (2010)
3. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
4. He, D., Arslan, A., Ling, A.: A fast algorithm for the constrained multiple sequence alignment problem. Acta Cybernetica 17(4), 701–717 (2006)
5. Khajeh-Saeed, A., Poole, S., Perot, J.: Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors. Journal of Computational Physics 229, 4247–4258 (2010)
6. Kloetzli, J., Strege, B., Decker, J., Olano, M.: Parallel longest common subsequence using graphics hardware. In: Favre, J., Ma, K., Weiskopf, D. (eds.) Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization. Eurographics Association (2008)
7. Ligocki, L., Rudnicki, W.: An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, pp. 1–8. IEEE Computer Society, Washington, USA (2009)
8. Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W.: GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, pp. 363–374. Springer, Heidelberg (2006)
9. Liu, W., Schmidt, B., Voss, G., Schroder, A., Müller-Wittig, W.: Bio-sequence database scanning on a GPU. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium, pp. 274–281 (2006)
10. Lu, C., Huang, Y.: A memory-efficient algorithm for multiple sequence alignment with constraints. Bioinformatics 21(1), 20–30 (2004)
11. Manavski, S., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. BMC Bioinformatics 9(suppl. 2), S10 (2008)
12. Munshi, A. (ed.): The OpenCL Specification. Khronos OpenCL Working Group (2010), `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`
13. NVidia Corporation: NVidia CUDA^{TM}Programming Guide, version 2.1 (August 12, 2008), `http://www.nvidia.com/object/cuda_get.html`
14. Schatz, M., Trapnell, C., Delcher, A., Varshney, A.: High-throughput sequence alignment using graphics processing units. BMC Bioinformatics 8(474), 1–10 (2007)
15. Suchard, M., Rambaut, A.: Many-core algorithms for statistical phylogenetics. Bioinformatics 25, 1370–1376 (2009)
16. Tang, C., Lu, C., Chang, M.T., Tsai, Y.T., Sun, Y.J., Chao, K.M., Chang, J.M., Chiou, Y.H., Wu, C.M., Chang, H.T., Chou, W.I.: Constrained multiple sequence alignment tool development and its application to RNase family alignment. In: Proceedings of the 1st IEEE Computer Society Bioinformatics Conference, pp. 127–137. IEEE Computer Society, Washington, USA (2002)