論文 / 著書情報
Article / Book Information

Title	A Multi GPU Read Alignment Algorithm with Model-based Performance Optimization
Author	Aleksandr Drozd, Naoya Maruyama, Satoshi Matsuoka
Journal/Book name	Springer's Lecture Notes in Computer Science N7851 (2012), vol. 7851, , pp. 270-277
発行日 / Issue date	2013, 1
DOI	http://dx.doi.org/10.1007/978-3-642-38718-0_27
権利情報 / Copyright	The original publication is available at www.springerlink.com.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

A Multi GPU Read Alignment Algorithm with Model-based Performance Optimization

Aleksandr Drozd[1] and Naoya Maruyama[1] and Satoshi Matsuoka[1]

Tokyo Institute of Technology, 2-12-1-W8-33, Ookayama, Meguro-ku, Tokyo 152-8552
(alex@smg.is.titech.ac.jp, naoya@matsulab.is.titech.ac.jp, matsu@is.titech.ac.jp)

Abstract. This paper describes a performance model for read alignment problem, one of the most computationally intensive tasks in bioinformatics. We adapted Burrows Wheeler transform based index to be used with GPUs to reduce overall memory footprint. A mathematical model of computation and communication costs was developed to find optimal memory partitioning for index and queries. Last we explored the possibility of using multiple GPUs to reduce data transfers and achieved super-linear speedup. Performance evaluation of experimental implementation supports our claims and shows more than 10fold performance gain per device.

1 Introduction

Faster and faster computing systems are developed every day to cope with ever-increasing complexity of problems that emerge in various areas of science and technology. Performance growth comes from technological advancements and mainly form architectures facilitating parallel data processing in various forms (i.e. recently GPUs). At the same time algorithms known to solve particular tasks themselves have many possibilities of improvement, taking into consideration fact that overall performance comes not just from better algorithm, but also on how it fits certain peculiarities of hardware platform and different patterns of data distribution in heterogeneous systems. GPUs and clusters of GPUs have recently become one of the main threads of supercomputing. Their computational characteristics are different from those of traditional systems and they are relatively new to software developers, which makes the above-stated issues even more important. Also while some applications have a pretty uniform data model, like those solving various matrix-based mathematical problems, in other applications data model itself is heterogeneous and its decomposition requires a profound study of balancing storage and distribution of workload parts so that we could better meet the platform characteristics and improve the overall performance.

This paper focuses on the pairwise local DNA sequence alignment problem. It is extremely computationally intensive as constant progress in sequencing technology leads to ever-increasing amounts of data to be processed. We target GPU-based systems that have been shown to allow for greater performance in sequence processing tasks due to their extreme parallel capacities [1].

Read alignment is basically a string matching problem and is typically done by building index of a reference and then matching queries against it. There are several types of indexes and corresponding match algorithms which were being used for alignment problem. We made a survey of existing solutions [2],[3],[4], and found that memory limitation is the performance bottleneck in all cases. Workload size for both reference sequence and query set can dramatically surpass available device memory and each index subdivision into smaller chunks to fit into memory simply doubles execution time. For example human genome contains approximately 3 billion of bases. Suffix array (array of integers giving the starting positions of suffixes of a string in lexicographical order) needs 9 bytes per base, so it will require 27 gigabytes of memory, while top modern GPUs have about 6GB. To index bigger references 64 bit integers are required and suffix array space complexity will be 17 bytes per base.

To reduce memory consumption we propose using matching algorithm based on Burrows Wheeler Transform. This algorithm is mainly used for data compression, but possibility of pattern matching using this transform was recently described[5]. Index based on BWT is more than ten times smaller than index based on suffix array. We perform an analysis of how this algorithm fits GPU characteristics and do model implementation to see if we can actually get significantly better execution time with this smaller memory footprint algorithm. This is the first contribution of this paper.

The second one is the performance model of possible memory utilization strategies. This model allowed us to find best proportions and succession of memory allocations and data transfers to maximize overall performance. We found that optimal performance is possible to achieve by using multiple GPU devices.

2 Background

In most living organisms the genetic instructions used in their development are stored in the long polymeric molecules called DNAs. To decipher this information we need to determine the order of nucleotides - the elementary building blocks of a DNA that are also called bases. This task is important for many emerging areas of science and medicine.

Modern sequencing techniques split the DNA molecule into pieces that are also called reads. Reads are processed separately to increase the sequencing throughput. Then they are aligned to the reference sequence to determine their position in the molecule. This process is called read alignment and is extremely computationally intensive, as a complete genome of such complex organisms as humans is billions of bases long, and the amount of reads data produced by sequencing machines is usually an order of magnitude bigger [6][7].

Technically read alignment is a substring matching operation: we search for a pattern of length m in reference string of length n, where $n >> m$. Straightforward naive approach has daunting asymptotic performance of $O(mn)$, so aligning is done by building index and than matching reads against it.

While theoretically fastest search algorithm uses suffix tree, its space complexity makes it inefficient for big references[8]. There were successful attempts to decrease memory footprint of matching algorithm or even to trade computational complexity for space consumption. In MummerGPU++ the authors replaced search algorithm based on suffix tree with one based on suffix array, which lead for another performance improvement[4].

Space complexity of suffix array is also linear, and constant multiplier under $O(n)$ is 9 bytes per symbol in case of two-bit implementation. Search complexity for suffix array is $O(m+\log n)$ where m is the length of query and n is the length of reference.

Evaluation of MummerGPU++ showed that on references over 100MB the memory limit is still taxing performance, since it leads to splitting the index into small pieces to fit into GPU memory and repeating search for each part. Search complexity does not depend (or depends very little) on index size, so splitting index in chunks increases computation time linearly. Copying index and queries to the device also takes its share of time of time. We will provide a more detailed analysis of time consumed by data transfers later on.

As the chief way to increase performance we propose using an algorithm with lesser memory footprint. Such an algorithm can be based on Burrows-Wheeler transform and some additional data structures (FM-Index) instead of suffix array. BWT was introduced in 1994 by Burrows and Wheeler[9] and was used mainly for data compression. There are some recent sequence alignment solutions using BWT, some of them are not parallel (Bowtie [10]), some are using GPUs, but for different class of alignment [11]. Also in [12] authors discuss the potential of using GPUs for exact sequence matching on single GPU.

3 BWT Based Aligner

The Burrows-Wheeler Transformation of a text T, BWT(T), is constructed as follows: The Burrows-Wheeler Matrix of T is the matrix whose rows are all distinct cyclic rotations of T$ sorted lexicographically. BWT(T) is the sequence of characters in the rightmost column of the matrix[9]. It is possible to use BWT for substring search. We adopted backward search algorithm proposed by Manzini and Ferragana [5] for GPU. Here Occ is the number of occurrences of given symbol before given position in transformed sequence. Array C contains total number of occurrences of each symbol.

BWT has a property called LF mapping: the i^{th} occurrence of character X in the last column of the BWT matrix corresponds to the same character in original text as the i^{th} occurrence of X in the first column. Backward_search procedure (fig. 1) uses LF mapping to calculate in rounds the rows of the matrix that begin with progressively longer suffixes of the query string.

The running time of the Backward_search procedure is dominated by the cost of evaluating $Occ(c, q)$. If we build a two-dimensional array OCC such that $OCC[c][q] = Occ(c, q)$ the backward search procedure runs in $O(m)$ time and it requires $O(|\Sigma|n \log n) = O(n \log n)$ bits.

The result of the Backward_search procedure is not the position(s) of matches in the reference sequence but the range of elements in the corresponding suffix array, containing indexes of actual matches in the reference. We suggest using suffix array on a host (which usually has enough memory to store it entirely) to decipher output of Backward_search procedure in $O(1)$ time. While it is possible to resolve positions of matches using the transformed text and OCC, generating all match positions on GPU will provide unpredictable amount of results per query, i.e. each execution thread will need to use unpredictable amount of device memory, and that is unsuitable for CUDA execution model. It will also cause additional overhead for moving data from device to host. To decipher search results on the host side we simply iterate suffix array elements bound by backward search procedure output values.

We use straightforward 2bits encoding for BWT itself. To compress OCC we split the transformed text into buckets of arbitrary size. For each bucket we will store the number of occurrences of each symbol in the transformed text before the first symbol of this bucket. For example, in 64 bit implementation for buckets of 32 symbols we will need 8 bits per symbol to store compressed OCC and 8 consequent memory reads to count the number of occurrences for any symbol. It gives us 10 bits of index per 8 bits of reference sequence and it is possible to change this ratio by varying OCC bucket size. 64 bit suffix array need 17bytes of memory, which is 13.5 times bigger. By merely replacing suffix array with BWT we already achieved 3-4 times performance improvement for cases where the size of data is too big to fit in memory for suffix-array based software but can be processed in one pass with our approach. Fig.2a) show how increasing reference size affects performance whether index can (BWT) or can not (suffix array) fit into GPU memory. We used NVIDIA Tesla 2050 card (2.6Gb memory) on the machine with 2.67GHz 4 cores Intel Core i7 920 CPU and 12GB of RAM running under CentOS 5.4.

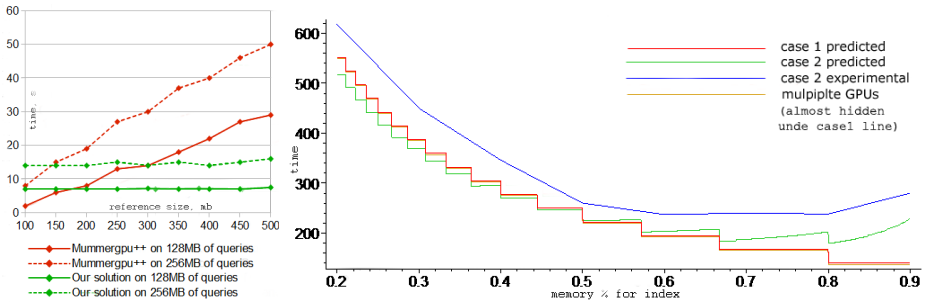
```
i:=p, c:=P[p],
First:=C[c]+1, Last:=C[c+1];
while ((First <= Last)
       and (i >= 2)) do
   c:=P[i-1];
   First:=C[c]+Occ(c,First-1)+1;
   Last:=C[c]+Occ(c,Last);
   i:=i-1;
   if (Last<First)
     then return no matches
     else return <First,Last>.
```

Fig. 1: Procedure Backward_search.

Experimental implementation takes reference and a set of named queries in FASTA format as input. Output is a set of positions in the reference where queries are mapped. We chose CUDA as target architecture as it is de facto standard for GPGPU programming. The algorithm was implemented in C++ for CUDA programming language.

The CUDA kernel that performs the query search is an almost straightforward implementation of procedure Backward_search, where each thread is processing its own query independently. Each thread stores results in its own preallocated global memory and accesses the reference index only by reading. Therefore there are no race conditions and no need for synchronization. Performance profiling showed that major share of time is consumed by loading data

from global memory. On references over 100mb MummerGPU++ starts to subdivide index and loses performance, while with our approach index up to several gigabytes (i.e. complete human genome) can be stored in GPU memory. For bigger reference sequence still must be subdivided. In the next chapter we present mathematical model of how memory partitioning affects performance and use it to find optimal parameters.



(a) Vs MummerGPU++ (b) Effect of memory partitioning

Fig. 2: Performance evaluation

4 Performance Model and Workload Balancing on single-GPU

The theoretical complexity of matching algorithm itself is $O(q)$, where q is query length. In case of sequential execution increasing number of queries to process obviously increases execution time in the same linear manner. So we can say that the overall execution time depends linearly on the overall size of query set. We just need to keep in mind need to have query set bigger than amount of data necessary to saturate GPU parallel capacity (which is in our case approximately 10mb, much is negligibly small).

Let us call memory size S_{mem}, index size S_{idx} and query set size S_{qry}. The overall execution time consists of the computation time itself and the time spent on moving data between host and device: $T = T_{cmp} + T_{mem}$. This formula assumes the worst case scenario when there is no overlapping between computation and data transfers. Cases where such overlapping is possible will be discusses below.

Suppose we have to split the index into N_{idx} chunks of size P_{idx} each and the query set into N_{qry} chunks of P_{qry} bytes. There is an obvious correlation between N_{idx} and N_{qry}, but for the time being we shall not include it in the model to keep it simpler. We have to match each chunk of query set against each part of index, one such iteration (kernel launch) taking $C * P_{qry}$ time as complexity

does not depend on index size. We have to repeat the matching procedure for each part of index and for each part of query set, which gives as execution time $T_{cmp} = C * N_{idx} * N_{qry} * P_{qry} = C * S_{qry} * N_{idx}$.

Now let's consider the communication expenses of moving index and query set parts from host to device. We have two basic options here. One option is to place one part of index on device, processing all subsets of query set one by one and then doing the same procedure for next part of index. The other option is to do the matching vice versa, i.e. matching one part of query set against all parts of index and then proceed to the next chunk of query set.

In the first case we need to copy P_{idx} bytes for each part of index, then N_{qry} times P_{qry} bytes of query subsets which equals to S_{qry} bytes and then to repeat this process N_{idx} times. Given host-to-device transfer bandwidth β communication will take $T_{mem} = \beta(P_{idx} + S_{qry}) * N_{idx} = \beta S_{idx} + \beta S_{qry} N_{idx}$ time. The overall time will be $T = C * S_{qry} * N_{idx} + \beta S_{idx} + \beta S_{qry} N_{idx} = (C + \beta) S_{qry} N_{idx} + \beta S_{idx}$.

For the second case using the same logic we get $T = C * S_{qry} * N_{idx} + \beta S_{qry} + \beta S_{idx} * N_{qry}$ overall execution time.

Let α be the share of memory occupied by index. Then each chunk of index will use αS_{mem} bytes and each chunk of queries $(1 - \alpha) S_{mem}$ bytes. We will have to split index into $N_{idx} = S_{idx}/\alpha S_{mem}$ chunks and query set into $N_{qry} = S_{qry}/(1-\alpha) S_{mem}$ chunks. Figure 2b shows how variation of α changes the overall execution time and that the first case allows for a potentially higher performance.

Actual value of C is retrieved form experiment and it depends on many parameters, like minimal required match length etc, but the asymptotic behavior will be the same. Performance of test implementation on big workloads confirms the predicted model (figure 2b).

So in the first case the overall performance increases as the index size is increased. This process continues up to the point where the memory remaining for queries is enough to run kernels with full memory saturation, which is relatively small and is not shown in figure2b.

In the second case we increase index size up until the point where communication expenses of repeating transfers of big index chunks are equal to the time spent on processing queries on extra number of index chunks. Maximal performance is better in the first case and it seems preferable from the point of view of pure GPU productivity. Moreover, it allows us to overlap communication and computation, as we can split queries without much penalty making performance even closer to ideal.

However, in this model we do not take into account the fact that results of matching of each subset of queries against each part or index need to be merged with each other. In the first case we have to store results of matching against each part of index somewhere until we process all queries and it will tax CPU-side memory/storage. This approach is completely inapplicable in a situation where queries are being streamed from some source (i.e. a sequencing machine) and we need to process each query block as it comes so we have to stay with worst case model - or we can try using multiple GPUs.

5 Multiple GPUs

Index chunk distribution among multiple GPU devices allows for smaller amount of repeatedly loaded index chunks per device. Ideally index chunks are not being moved at all. In this case theoretical performance in terms of pure GPU productivity will be even better, though not significantly, than that provided by the first approach on a single GPU device. On each device we spend $C * S_{qry} + \beta S_{qry}$ time for moving and processing all queries (once again, overlapping is possible in this case).

The process of deciphering and joining results consists of following stages. We get the ranges of suffix array elements as output of each GPU matching routine and restore actual positions of matches in reference sequence. For each device output we will have such list of positions. Then we need to merge these lists together and sort resulting list. It does indeed look like time consuming routine, but it obviously has $O(N_{idx})$ complexity, the same as complexity of search procedure itself. The exact multiplier depends on implementation, CPU characteristics and average number of matches for each query. However, given realistic search output, our sequential test implementation performed merging of 8 chunks of one million results in less then one second, which is definitely faster than processing corresponding amount of data on GPU (fig. 3). In previous experiments we used queries of 100 bases long, so 1 million results correspond to 100Mb of query data. In tests on both real and generated sequences multi GPU performance per device was same as for single GPU case 1. We performed benchmarking on one of the Tsubame 2.0 supercomputer nodes with 2 six-core Intel Xeon X5670 CPUs and 54GB of RAM running under SUSE Linux Enterprise Server 11 SP1 for this test. The node has three NVIDIA Tesla 2050 GPUs connected with 16 lanes of PCI Expression 2 on it. We used 100 bases long queries and set minimal match length to 40 bases. For 6GB reference sequence aligning efficiency per device was 3.55 million bases per second for single GPU and 3.7 for multi GPU implementation when all 3 devices were used. So 3 GPUs compared to single one gave us 3.11 times speed-up, i.e. 1.04 efficiency. Optimal number of devices is equal to the number of index chunks of optimal size. Increasing number of GPUs further will negatively affect the efficiency as index chunk size will be decreased.

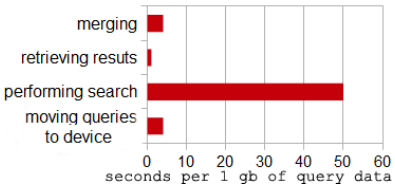


Fig. 3: Performance details

6 Conclusion

Better software performance does not necessarily come from computational complexity of underlying algorithms. Choice of particular data structures and corresponding algorithms depends on how they meet characteristics and features of target hardware. This is particularly true for GPU devices.

This paper shows that using more compact data structures can lead to performance improvement in short read alignment problem. We refactored MummerGPU++, previous highly-efficient GPU exact-matching read alignment software by replacing suffix array with BWT and rewriting the corresponding search algorithms and get 3-4 times performance improvement. The analysis of application behavior for the case of workload size considerably exceeding device memory proves that higher performance can me achieved by intelligent strategy for data decomposition. We also showed that best performance per device for read alignment problem can be achieved by using multiple GPUs, and the optimal number of GPU devices for a particular task can be estimated from reference size.

References

1. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohm, and T. J.Purcell, "A survay on general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.
2. A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic Acids Res.*, vol. 27, p. 2369, 1999.
3. M. C. Schatz, C. Trapnell, A. L. Delcher1, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, p. 474, 2007.
4. A. Gharaibeh and M. Ripeanu, "Size matters: Space/time tradeoffs to improve gpgpu applications performance," in *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010.
5. P. Ferragina and G. Manzini, "Indexing compressed text." *Journal of the ACM*, vol. 53, no. 4, pp. 552–581, 2005.
6. M. Pop, "Genome assembly reborn: recent computational challenges," *Briefings in Bioinformatics*, vol. 10, p. 354, 2009.
7. J. M. Rothberg, W. Hinz, T. M. Rearick *et al.*, "An integrated semiconductor device enabling non-optical genome sequencing," *Nature.*, no. 475, pp. 348–352, 2011.
8. D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
9. M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Technical Report 124, 1994.
10. B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome." *Genome Biology 10 (3).*, vol. 10, no. 25, 2009.
11. R. Li, C. Yu, Y. Li *et al.*, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 15, no. 25, pp. 1966–1967, 2009.
12. S. Chen and H. Jiang, "An exact matching approach for high throughput sequencing based on bwt and gpus," in *2011 IEEE 14th International Conference on Computational Science and Engineering (CSE).* IEEE Computer Society, 2011.