Vol.2011-HPC-130 No.13
2011/7/27

Fast GPU Read Alignmennt with Burrows Wheeler Transform Based Index

Aleksandr Drozd ,†1 Naoya Maruyama †1
and Satoshi Matsuoka†1

This paper addresses the problem of performing faster read alignment on GPU devices. The computationally-intensive task of DNA sequence processing is approached from the perspective of parallel computation. We found memory limitations to be one of the biggest performance issues, and attempted to decrease memory footprint of alignment algorithm to boost GPU performance. Suggested implementation uses index based on Burrows-Wheeler transform and shows 3-4 time speed improvement over the previous fastest solution.

1. Introduction

The problem of DNA sequence processing is extremely computationally intensive as constant progress in sequencing technology leads to everincreasing amounts of sequence data. One of possible solutions for this problem is using the extreme parallel capacities of modern GPU devices[8)]. However, performance characteristics and programming models for GPU differ from those of traditional architectures and require new approaches.

GPU devices outperform traditional processors due to their parallel capacity, but this parallelism is available in single-instruction multiple-thread form. Most importantly, host memory and I/O systems are not directly accessible from a GPU device and onboard GPU memory is usually an order of magnitude smaller that memory size on a host. Considering the size of read alignment data the memory limit becomes a real problem: when reference sequence index does not fit into memory it has to be split into parts that would be processed individually. In most cases the complexity of the algorithm does not depend on the index size, or the dependence is not linear, but logarithmic. Splitting of index increases computation time tremendously.

Attempts have been made to decrease memory consumption of the matching algorithm[4)], and we suggest to make another step in this direction by building a much smaller index using Burrows-Wheeler Transform (BWT). At the same time we will continue using suffix array on host side to benefit from computational characteristics of both GPU and CPU. We reduced index size 12 times and achieved 3-4 time performance improvement.

2. Problem Domain

In most living organisms genetic instructions used in their development are stored in the long polymeric molecule called DNA. 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 must be 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 produces by sequencing machines is usually an order of magnitude bigger. Moreover, constant progress in sequencing technology provides more and more data output per time[9)].

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 typically matching is done in two stages:

- Index is build from the reference DNA sequence;
- Each read is matched against the reference sequence using its index.

Several existing solutions that use different types of search index are briefly discussed in the following section.

†1 Tokyo Institute of Technology

© 2011 Information Processing Society of Japan

3. Related Work

The theoretically fastest search algorithm uses suffix tree as index and has computational complexity $O(m)$ (where m is query length) for matching one query to the reference. Also in read alignment we usually search for the longest possible match up to some minimal match length. Instead of repeating search for each subquery the suffix tree can incorporate additional links that connects related suffixes. Thus it allows to search for all subqueries of a given query in $O(m)$ time

MummerGPU read alignment software was developed based on this data structure[2]. Later on it was refactored into GPU-version, and its authors claimed 3-4 time speed-ups over the CPU version[10].

While the suffix tree asymptotic space complexity is linear, the constant multiplier under $O(N)$ (where N is reference length) is very big, between 22.4n and 32.7n bytes for DNA sequences[6], so the memory consumption becomes a serious performance issue on big workloads.

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].

Suffix array is simply an array of integers giving the starting positions of suffixes of a string in lexicographical order[7]. Space complexity of suffix array is also linear, and constant multiplier under $O(n)$ is 9 bytes per symbol in case of 32bits 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 workloads over 100mb of reference size 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 each iteration increases computation time linearly. Copying index and queries to the device also takes a lot of time.

4. Our Solution

We propose using index based on Burrows-Wheeler transform and some additional data structures (FM-Index) to get more performance from the GPU. BWT was introduced in 1994 by Burrows and Wheeler[1] and was used mainly in compression algorithms such as bzip2 as it transforms reoccurring patterns in the string into continuous runs of a single symbol.

4.1 Performing Search with BWT

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$ that are sorted lexicographically. BWT(T) is the sequence of characters in the rightmost column of the matrix. Manzini and Ferragana showed that BWT can also be used as efficient search index[3], even while preserving certain compression.

BWT has a property called called LF mapping: the ith occurrence of character X in the last column of the BWT matrix corresponds to the same character in original text as the ith occurrence of X in the first column.

Backward_search procedure (figure 1) uses LF mapping to calculate in rounds the rows of the matrix that begin with progressively longer suffixes of the query string.

```
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.

Here Occ is the number of occurrences of a particular symbol before the symbol in a given position of BWT. Array C contains the number of ocurrences of each symbol in whole text.

© 2011 Information Processing Society of Japan

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 will require $O(|\Sigma| n \log n) = O(n \log n)$ 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.

It is possible to resolve positions of matches using the transformed text and OCC, but generating all the match positions in GPU will provide unpredictable amount of results per query and this will slow down the kernel because threads will have to compete for GPU memory. It will also cause additional overhead for moving data from device to host. So 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.

4.2 Compressing BWT

We use the fact that DNA sequences has a very small alphabet (four symbols), and apply straightforward encoding using two bit for each symbol.

Such compression is almost as efficient as bzip-like. Other benefits include the absence of worst-case degradation and the possibility to estimate memory space required for index before the transform. With this approach we can also split the reference sequence into chunks that would fit the available memory or distribute it equally between several GPU devices.

4.3 Storing OCC

We will split 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, for buckets of 32 symbols we will need 4 bits per symbol to store OCC and 8 consequent memory reads to count OCC.

Thus to store the whole index for reference sequence containing n bases we shall need 6n bits, which is 12 times smaller than for suffix array (72bits).

5. Implementation

To preserve compatibility with previous implementations such as Mummer, MummerGPU and MummerGPU++, we use the same input and output format. The program takes reference and a set of named queries in FASTA format as input. Output is a set of queries with the positions in the reference where they are mapped and the position of characters that match the reference in each query.

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 program executes in following phases:

(1) Build index from reference or load pre-built index.
(2) Load query set.
(3) Move index and queries to GPU.
(4) Align queries to reference using its index.
(5) Copy results to device.
(6) Print results.

Phase 4 is the only one that runs on GPU, and it is also the performance bottleneck of the whole program.

The CUDA kernel that performs the query search is an almost straight-forward implementation of procedure Backward_search, where each tread 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.

Unfortunately the nature of algorithm presupposes memory reads from random and unpredictable locations, so l1 cache performance counters show that 60% of memory reads lead to cache miss. Data transfer between host and device takes less than 10% of computation time and is no longer a performance bottleneck as is the case with previous solutions.

5.1 Performance Evaluation

We performed benchmarking on a system with Nvidia Tesla C2050 cards (2.6 Gb memory, compute capability 2.0) on both real data as well as generated sequences to see performance dynamics on different amounts of data. Aligning a million of reads 100 bases long to the human chromosome 1 (380gb) took approximately 10 seconds. Figure 2 shows speed improvement dynamics over MummerGPU++ as reference size is increased.

© 2011 Information Processing Society of Japan

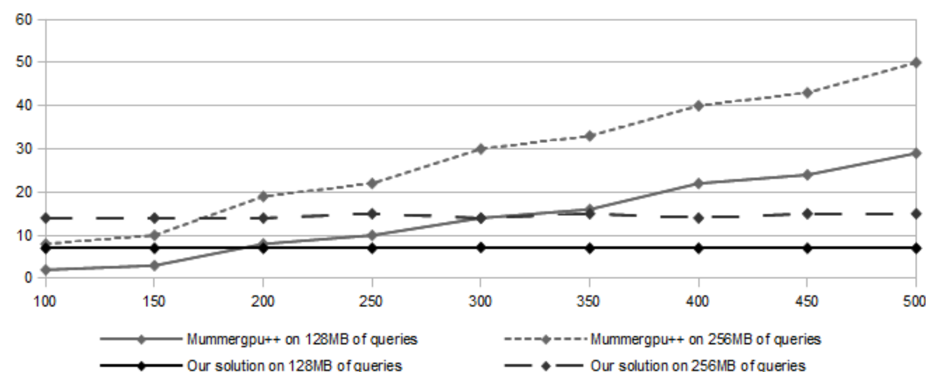

Fig. 2 Comparative performance diagram (kernel time)

Due to these microarchitectural issues the actual performance improvement is not as big as can be predicted theoretically by analysis of the algorithm, but still considerable. The possibility of further optimisation may be in use of more aggressive compression of the transformed text and OCC array (E.g. Ferragana and Manzini[3] suggest applying move to front, runlength and arithmetic coding subsequently to transformed text, preserving the possibility for conducting search).

6. Conclusions

Better software performance does not necessarily come from computational complexity of underlying algorithms. Choice of a particular data structure and a corresponding algorithm depends on how they meet characteristics and features of target hardware, and that is especially true for GPU devices. This paper shows that using more compact data structures can lead to a performance improvement in short read alignment problem, and our tests on implemented BWT-based algorithm agree with theoretical predictions. We refactored Mummerpgu++, previous fastest exact-matching read alignment implementation on GPU, by replacing suffix array with BWT and rewriting the corresponding search algorithms. Evaluation on Nvidia Tesla C2050 device showed 3-4 times performance improvement over MummerGPU++. There are several sequential implementations of alignment software using Burrows-Wheeler transform as search index. Software called Bowtie combines BWT with backtracking algorithm to allow for approximate read alignment[5]. We can not make direct comparison with this solution as it implements a different class of alignment, while we currently focus on exact matching. However, many approximate matching algorithms can be based on BWT and FM-Index, and we hope to evaluate these algorithms on GPU in future.

References

1) Burrows, M. and Wheeler, D.J.: A block-sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation (1994).
2) Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O. and Salzberg, S.L.: Alignment of whole genomes, *Nucleic Acids Res.*, Vol.27, p.2369 (1999).
3) Ferragina, P. and Manzini, G.: Indexing Compressed Text., *Journal of the ACM*, Vol.53, No.4, pp.552–581 (2005).
4) Gharaibeh, A. and Ripeanu, M.: Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance, *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society (2010).
5) Langmead, B., Trapnell, C., Pop, M. and Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome., *Genome Biology 10 (3).*, Vol.10, No.25, pp.45–60 (2009).
6) M.I.Abouelhoda, S.K. and Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays, *Journal of Discrete Algorithms*, Vol.2, pp.53–86 (2004).
7) Manber, U. and Myers, G.: Suffix arrays: A new method for on-line string searches, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pp.319–327 (1990).
8) Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohm, A.E. and J.Purcell, T.: A Survay on General-Purpose Computation on Graphics hardware, *Computer Graphics Forum*, Vol.26, No.1, pp.80–113 (2007).
9) Pop, M.: Genome assembly reborn: recent computational challenges, *Briefings in Bioinformatics*, Vol.10, p.354 (2009).
10) Schatz, M.C., Trapnell, C., Delcher1, A.L. and Varshney, A.: High-throughput sequence alignment using Graphics Processing Units, *BMC Bioinformatics*, Vol.8, p.474 (2007).

© 2011 Information Processing Society of Japan