Accelerating SOLiD short read assembly with GPU

Péter Szántó, Béla Fehér
Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
{szanto, feher}@mit.bme.hu

András Széll
FPGArt Ltd.
Budakeszi, Hungary
szell@fpgart.hu

***Abstract*—High-throughput genome sequencers recently became available to the masses what greatly increases the number of laboratories using these machines. Unlike the traditional Sanger sequencing, these sequencers uses much shorter reads and generate a large amount of data. Processing the sequencer output with pure software methods often leads to unacceptable run-times; therefore some kind of hardware acceleration is extremely beneficial.**

I. Introduction

The genome contains all hereditary information of a living being, which includes genes and non-coding sequences. From computational point of view, the genome is nothing more than a list of so-called base pairs (bp), namely A (adenine), C (cytosine), G (guanine) and T (thiamine). The number of base-pairs in the genome is peculiar to a given organism – e.g. the genome of a simpler virus contains thousands of base-pairs, while the human genome has more than 3 billion pairs. Genome sequencers generate a large number of genome fractions as their output. Sanger sequencing technology produced ~1000bp – in contrast, the recently developed high-throughput sequencers generate much shorter genome fractions (called reads) in the range of 30-40bp which are called short-reads. Such sequencers are manufactured by Illumina, Applied Biosystems and Helicos. As the reads are very short, sequencing is done by “mapping” the reads to a reference genome – this type of assembly is often referred as comparative assembly or resequencing.

Most sequencers produce the reads in letter-space with the exception of Applied Biosystems’ Solid technology, which generates so called color-space reads.

II. Color space short reads

During the sequencing process, Solid technology do not identify the individual bases, but instead it uses 2-base-encoding to encode two-base combinations. During the sequencing four fluorescent dyes are used to encode the sixteen possible two-base combinations. That is, the difference between two consecutive bases is encoded into single color information, as Figure 1. shows.

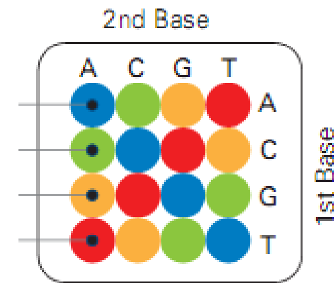


Figure 1. Solid two base color encoding

To be able to precisely decode color-space reads into letter-space, the leading base has to be defined (the first color defines the difference in contrast with this leading base). As all available samples use T as a leading base, we also choose this for our application.

Reference genomes are available in letter-space, so they contain A, C, G and T characters. Mapping Solid’s color-space reads to the letter-space reference genome can be done in color-space or in letter-space. As Figure 2. shows, if the color-space data is converted into letter-space, a single color-space sequencing error invalidates all letters after the position of the error. If mapping is done in color-space, a single sequencing error remains only a single error.

Figure 2. Single sequencing error

Therefore, our application does the read mapping in color space. To be able to do this, the reference genome is converted into color space before the mapping process.

III. Proposed mapping algorithm

Basically, mapping is a simple process: all the reference genome positions should be found where the sequenced read

fits with little differences. When mapping genome sequences it is not adequate to compute perfect matches only for two reasons. First, the sequencer output may contain errors. Second, one goal of the genome sequencing is to compare individuals' genomes to find small differences which can help identify the genetic reasons of hereditary diseases. Therefore, some differing bases should be allowed during the mapping – for the 25bp Solid reads 3-4 differences are adequate. It should be noted that the reads may contain several types of errors, the most complex ones (like insertions or deletions) are not supported by our software in the current phase.

The basic mapping process is the following. Reads should be slid through the whole reference genome and those reference positions should be saved where the difference is smaller than a predefined threshold (mismatch – number of differences allowed). As the length of reference human genome is ~3 billion bp, this means that a single read should be compared with ~3 billion reference fractions which are as long as the read itself. The number of reads generated by the sequencer (with ten times coverage) is ~120 million. Therefore, the number of required comparisons with a brute-force method is 120 million multiplied with 3 billion – this leads to unacceptable run times. To reduce the number of operations required the proposed method employs hashing. Although current Solid reads are 25 bp long, our solution is prepared for longer read length.

The algorithm has six main steps:

1. Load the reads and the reference genome. Internally – irrespectively of the actual input file format – both color-space and letter-space bases are stored on two bits.
2. encode the reference into color-space (generate read-long, overlapping fractions)
3. hash the reads and the reference fractions
4. generate inputs for the accelerator; start processing; process accelerator output
5. re-run steps 3. and 4 for modified hash (if necessary)
6. merge outputs, write results

As the number of comparisons is reduced by the hashing, selecting optimal hashing function is crucial.

A. *Binning input data*

Hashing groups reads and reference fractions into bins by masking several bits and then group the inputs based on the masked value of the read/reference. The number of bins generated therefore equals to 2^{mask_bits}. During the actual comparison only the corresponding read and reference bins are processed (that is, read bin 0 is only compared with reference bin 0, and so on). Thus, using a single mask, differences in the bits used for masking are not allowed. To find read-reference mappings which has at most mismatch differences in those bits used for masking, another mask has to be used. The number of masks in a mask-set depends on the number of bits used for masking and the mismatch value. A good mask-set therefore contains the fewest possible masks which still offers full coverage (that is all possible mismatch combinations could be detected with at least one mask from the set). The proposed implementation uses mask sets from the La Jolla Covering Repository [2]. Figure 3 shows a mask set which uses 8 bases (16 bits) and allows three mismatches (yellow boxes denote the bases used for masking).

Mask 0	1	2	3	**4**	**5**	6	7	8	9	**10**	**11**	12	13	14	15	**16**	**17**	18	19	20	21	**22**	**23**	24	25
Mask 1	1	2	**3**	4	**5**	6	7	8	**9**	10	**11**	12	13	14	**15**	16	**17**	18	19	20	**21**	22	**23**	24	25
Mask 2	1	2	**3**	**4**	5	6	7	8	**9**	**10**	11	12	13	14	**15**	**16**	17	18	19	20	**21**	**22**	23	24	25
Mask 3	1	**2**	3	4	5	**6**	7	**8**	9	10	11	**12**	13	**14**	15	16	17	**18**	19	**20**	21	22	23	**24**	25
Mask 4	**1**	2	3	4	5	**6**	**7**	8	9	10	11	**12**	**13**	14	15	16	17	**18**	**19**	20	21	22	23	**24**	25
Mask 5	**1**	**2**	3	4	5	6	**7**	**8**	9	10	11	12	**13**	**14**	15	16	17	18	**19**	**20**	21	22	23	24	25

Figure 3. Mask set example

The drawback of such mask-sets is that the same result may be generated with different masks and these duplicates should be removed afterwards (for example if a read and a reference has difference only at base positions {1, 2, 6} this result is generated with masks 0, 1 and 2). Table 1. shows the number of required masks within a mask set for different mask bit sizes and mismatch values (M), assuming 25 base reads.

	M=0	M=1	M=2	M=3	M=4
16 bit	1	2	3	6	11
20 bit	1	2	4	10	21
24 bit	1	2	6	14	30
28 bit	1	2	7	23	66

Table 1. Number of masks in a mask set

The implemented hashing algorithm generates block-based linked list as an output. That is, at the beginning every bin has a relatively small memory block allocated to it (16 reads or references). If necessary, a new block is allocated and chained to the previous one. According to our measurements, this type of bin generation is considerably faster than an in-place solution. The drawback of this method is memory requirement: for every bin the last allocated block may not be fully utilized; with 16bit masks ~8Mbyte of memory may be wasted, but with 24bit masks wasted memory size may be as high as 2 Gbytes.

B. *Input data for the accelerator*

After hashing, the host software generates inputs for the accelerator, namely a command stream and a data stream.

The data stream contains reads and reference fractions from several bins in 128-bit format, which allows 64 base inputs to be processed. All data is merged into a continuous memory area.

The command streams store commands which should be executed on the data stream. More precisely, a single command contains the following information:

- offset in the data stream of the reads to be compared
- number of reads to be processed
- offset in the data streams of the references to be compared
- number of references to be processed
- maximal allowed mismatch value

IV. GPU SOFTWARE

The actual read – reference comparisons are accelerated by an NVIDIA GTX260 GPU. The GPU is programmed in NVIDIA's C-based CUDA programming language [3].

The host CPU copies both the command stream and the data stream into the on-board memory of the GPU. In the case of GPU acceleration, the number of reads to be processed is 128 at the most for each command. If a read bin contains more than 128 elements, multiple commands are generated. Unlike reads, the number of references in the command is only limited by the available memory.

The reason behind generating 128-read commands is that a single command is processed by a single thread-block on the GPU. The number of thread blocks in a single run equals to the number of commands generated. The GPU software does the following steps:

1. Read the command associated with the given thread block. E.g. thread block 0 reads command 0; thread block 1 reads command 1; and so on.
2. Every thread loads its assigned read into a GPU register.
3. If not all input references were processed, every thread block reads 128 references into shared memory. Therefore, all threads within a thread block read a single reference.
4. All threads within the thread blocks iterate through the 128 references and compare the reference with its own read.
5. If all necessary conditions come true, the index of the read (which is actually the ID of the thread) and the index of the reference are written to the on-board memory.
6. Steps 3-6 are repeated as long as necessary.

One crucial point in getting the most performance out of the GPU is memory handling. Although the bandwidth of the on-board memory itself is considerably higher than the system memory bandwidth of CPUs, it still can easily be a limiting factor because of the large number of processing cores. NVIDIA GPUs offer on-chip shared memory which is shared between threads inside a thread block. One notably important feature of this type of memory is broadcasting, whereby the same data is read from the memory with a single read request and broadcasted to several threads.

Due to the way commands are generated, reads processed within a thread-block belong to the same read bin, therefore they have to be compared with the same references. Thus, if threads are synchronized before shared memory read (so that they read the same shared memory address at the same time) the effective read bandwidth multiplies by the broadcasting feature.

Another property of the shared memory is that it contains multiple banks which can be accessed simultaneously if there is no bank conflict between the threads. Writing the reference data into shared memory is done in a way that this property can be exploited.

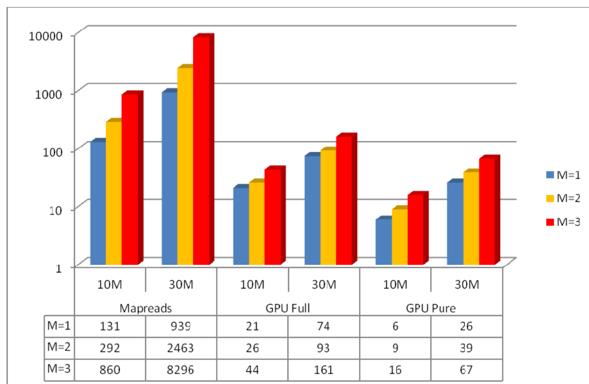
Performance is decreased if the threads within a so called thread warp execute divergent branches, so to avoid this situation all threads are always executed, even if the read associated with the thread is invalid – this is the case when the command of the thread block contains less than 128 reads. However, threads processing invalid reads are not allowed to write to the result memory.

The number of matching references are unknown before the comparison, therefore the GPU employs block-based linked list to generate output data. As an input, the GPU receives the base address of the first input block for every thread and the address of the first free block. If a thread needs a new memory block it, it allocates it by reading the address of the first free block and updating that address. To ensure that no parallel block allocation happens this requires atomic operation which is available in devices with computing capability 1.1 and higher. At the end of the processing, this address shows the number of allocated blocks which corresponds with the amount of output data generated.

V. RESULTS

For the development, CUDA SDK 3.0beta and Visual Studio 8.0 were used on a standard desktop system (Core2 Duo CPU @3.8 GHz, 8Gbytes of memory).

In the figure below two results are shown in the GPU accelerated case: the full run-time (GPU Full) and the run-time of GPU related functions, which includes data movement to/from the graphics card and GPU kernel execution time (GPU Pure). Our solution is compared to the results obtained with Applied Biosystems' mapreads software. Tests were ran with two input databases: in the first case 10 million reads and references; in the second case 30 million reads and references. The maximal allowed mismatch (M) were set to 1, 2 and 3; in all cases 8 bases were used to bin input data. It should be also noted that – unlike mapreads – our solution produces all results, there is no limit on the number of outputs.



	Mapreads 10M	Mapreads 30M	GPU Full 10M	GPU Full 30M	GPU Pure 10M	GPU Pure 30M
M=1	131	939	21	74	6	26
M=2	292	2463	26	93	9	39
M=3	860	8296	44	161	16	67

Figure 4. Run-time results (seconds)

The preliminary results shown on Figure 4. are promising, the accelerated version handles increasing data sizes much better than mapreads. The full human genome is several times larger than our test databases, so the better scaling of the GPU version could be a great advantage.

The presented solution is more a proof of concept than a fully optimized version, there is potential for further acceleration. For example, in the current version the thread group size is a compile-time constant – with hierarchical binning and more adaptive thread management the GPU occupancy increased. The performance of the host software can be further increased with support for more threads – as our development platform has two processor cores, currently two threads are used.

REFERENCES

[1] Applied Biosystem SOLiD documentation, solid.appliedbiosystems.com

[2] La Jolla Covering Repository, http://www.ccrwest.org/cover.html

[3] NVIDIA CUDA documentation, http://www.nvidia.com/object/cuda_home_new.html

[4] Next Generation Genome Sequencing, Dr. Michal Janitz

.