S8130

Building a GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

Richard Wilton
Department of Physics and Astronomy
Johns Hopkins University

Thank you...

I'd like to start by thanking my colleagues at Johns Hopkins: Xin Li, Andy Feinberg, and Alex Szalay.

A few years ago, I was involved in building a GPU-accelerated short-read aligner called Arioc.

Here is what a short-read aligner does: A DNA sequencer does not process a sample of DNA by reporting its sequence from end to end. Instead, it generates hundreds of millions or billions of short DNA sequences which we call "reads". We then use software like Arioc to figure out where each of those reads might have come from in the original DNA by comparing each read's sequence to a normal reference sequence.

It turns out that Arioc is very good at rapidly finding alignments for short DNA reads.

There is, however, a great deal of interest in being able to align bisulfite-treated DNA sequences – and because the different chemistry involved in generating bisulfite-treated short reads changes the way the read sequences are interpreted, the read aligner needs to some additional work. A read aligner that has not been designed for this specific task cannot handle bisulfite-treated DNA reads.

We already had Arioc, so we decided to add the ability to handle bisulfite-treated DNA sequences to the existing Arioc implementation. What I'm about to show you is how we approached the problem of making this happen on the GPU.

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

Short-read alignment

Perfect alignment

```
R: CATGTGTGAAGCCTCCATACTTGAGTCCTGAACTGATGAACTAA
           ||||||||||||||||||||||||||||||
Q:         AAGCCTCCATACTTGAGTCCTGAACTGATGAA
```

Alignment with mismatches

```
R: CATGTGTGAAGCCTCCATACTTGAGTCATGAACTGATGAACTAA
           |||||||||||| ||||||| ||||||||||||
Q:         AAGCCTCCATACCTGAGTCCTGAACTGATGAA
```

Alignment with mismatches and gaps

```
R: CATGTGTGAAGCCGCGCGTCCATACTTGAGTCATGAAC--ATGAACTAA
           ||||||      |||||| |||||| |||||   |||||
Q:         AAGCCT-----CCATACCTGAGTCCTGAACTGATGAA
```

Scoring example

Parameters	
match	+2
mismatch	−6
gap	−5
space	−3

↓

Scores	
perfect	64
mismatches	48
mismatches and gaps	11

When you sequence an individual's DNA, you basically chop the DNA into millions or billions of short pieces and run them through an automated chemical process that identifies the sequence of chemical building blocks of each piece. This happens in an apparatus called a DNA sequencer, and takes a day or more to accomplish (although modern sequencers amortize that by processing multiple DNA samples concurrently).

The sequencer's output is a billion or more character strings, which we informally call "reads". Each read contains one character per chemical building block. The central problem in short-read alignment is to figure out where those reads came from in the original DNA, which can be 7 orders of magnitude longer than a read.

To do that, we use a "reference sequence" whose sequence represents a statistically-valid expectation of what a normal individual's DNA looks like. So a short-read aligner is basically a software tool for doing inexact string matching between hundreds of millions or billions of short (100-250 symbol) strings and a single long string that can contain billions of symbols (in the case of human DNA, it's about 3 billion).

In this slide, R is the reference sequence, or more informally, the "genome"; Q is one of the short reads emitted by the DNA sequencer.

You can see three different ways a read can be mapped to the genome: perfectly, with one or more mismatches, and with gaps. The read aligner assigns a score to each mapping based on a simple scoring system, so for the 32-character reads in this example, the alignment scores would be

64: a perfect mapping, scored at 2 points per matching symbol
48: a mapping with 30 matching symbols = 60, plus 2 mismatched symbols = -12
11: a mapping with 27 matching symbols = 54, plus 2 mismatched symbols = -12 , plus 2 gaps = -10, plus 7 gap spaces = -21

The algorithms that do this kind of string alignment are neither pretty nor fast – which is why for the past 10 years or so, people have been trying to use GPUs to accelerate short-read alignment computations.

Short-read alignment on GPUs

- The two tough problems associated with GPU-accelerated short-read alignment
  - Not amenable to cooperative multithreaded implementation
  - Memory-intensive
- GPU-accelerated implementations
  - SOAP3-DP (2013)
  - NVBIO (2015)
  - Arioc (2015)

The two big problems with GPU-accelerated short-read alignment have to do with the nature of the algorithms we have for inexact string matching.

- None of the basic read-alignment algorithms are easy to parallelize by using multiple cooperative CUDA threads. You're almost always better off using one CUDA thread to compute an alignment on each read.
- You need a copy of the reference sequence to compute alignments. You also need some kind of index structure or lookup table to figure out where in the reference sequence to do the alignment computations. These data structures consume a big chunk of GPU memory. It's also very hard to access them efficiently with CUDA coalesced memory techniques.

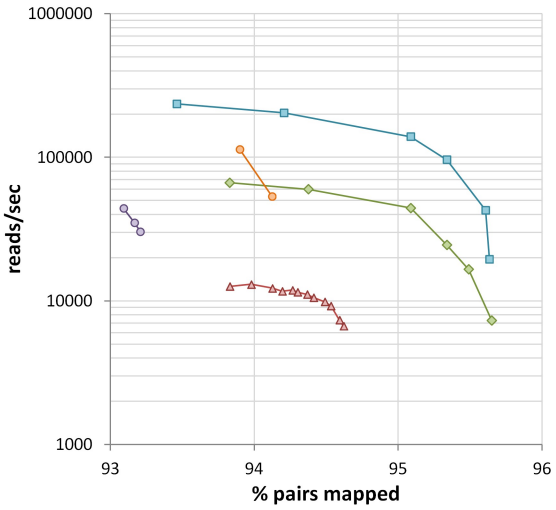
There are three published GPU-accelerated implementations that provide accuracy comparable to the most widely used CPU-based programs:

- SOAP3-DP was developed at the University of Hong Kong
- NVBIO is a product of an Nvidia research team
- Arioc was built by our own group at Johns Hopkins

There are several other experimental GPU-accelerated implementations out there, but they do not offer much speedup compared with CPU-only implementations, so I haven't mentioned them here.

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

10x



- Multithreaded CPU implementations
  - BWA-MEM
  - Bowtie 2
- GPU-accelerated implementations
  - SOAP3-DP
  - NVBIO
  - Arioc

- Workstation configuration
  - 12 CPU cores (24 threads) at 2.93GHz
  - One Nvidia K20c GPU
  - Arioc

In our GPU development, we have a rule of thumb that a successful GPU-accelerated implementation is at least ten times faster than the comparable multithreaded CPU-based version.

With this in mind, here are some performance results for the fastest CPU- and GPU-based short-read aligners. These data are a few years old, as you can tell from the hardware, but they have held up pretty well with newer software versions and more highly parallel hardware.

What is important here is the tradeoff between sensitivity and speed. Most short reads are easy to map in that their sequence is specific to very few locations in the reference genome, so the read aligner only computes a few potential alignments in order to find the best ones.

There is, however, a small percentage of reads that are hard to align, either because they have potential mappings at many different locations in the reference genome or because their sequences aren't that similar to anything in the reference. In both cases, a read aligner may need to compute hundreds or thousands of alignments for a read in order to find mappings with high-enough scores to report. This accounts for the logarithmic drop-off in speed as the aligner's sensitivity increases.

In any event, you can see that for comparable sensitivity settings, GPU-accelerated short-read aligners can achieve a ten-fold speedup when compared to their multithreaded CPU-only counterparts.

[It also shows you how hard it its to interpret the vast majority of published speed results for short-read aligners, since there is no one "speed" number you can use to describe an aligner's performance. But let's not go there…]

The question is: can we accomplish the same thing for a slightly different read-alignment problem?

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

Aligning BS-seq reads

- Bisulfite-treated DNA sequencing (BS-seq) is used to identify methylcytosine (C_m) in the DNA of higher organisms
- How do you detect methylcytosine in DNA reads?
  - There is no high-throughput chemical pipeline capable of identifying C_m in a DNA sequencer
  - Bisulfite (HSO_3^-) treatment of DNA converts C (but not C_m) to T
  - The aligner infers C, T, or C_m in each mapped read by comparing it with the reference

C-T ambiguity

In untreated DNA	In sequencer reads
A	A
C_m (methylcytosine)	C
G	G
C or T	T

Resolving the ambiguity

Reference	AGTCC
Read	AGTTC
Inferred sample sequence	AGTCC$_m$

Now we now turn to the problem of aligning DNA sequencer reads that contain methylcytosine (C_m) in addition to A, C, G, and T (the four DNA bases everybody learns about in elementary school).

Methylcytosine is chemically similar to cytosine. It has not yet been possible to develop a DNA sequencer protocol that will reliably distinguish one from the other. So a biochemical trick is used instead: the DNA sample is treated with bisulfite (HSO_3^-), which converts C (but not C_m) to T in the read. (Actually the chemistry is more complicated than that, but that doesn't matter here.)

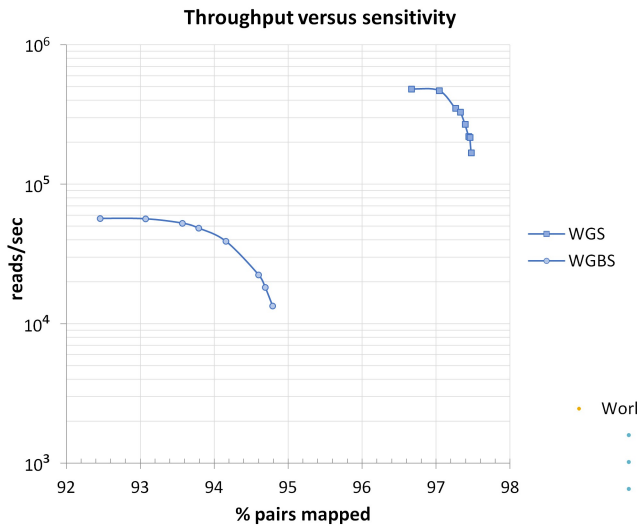
The sequencer knows about T and it treats C_m as C, so it reports read sequences that are full of Ts. The only Cs in the reads are found where a C_m existed in the original read sequence.

As you can see from the table, the read aligner must disambiguate Ts in the reads. It does that after it finds a mapping for each read. For each T in the read, the aligner looks at the corresponding position in the reference. If the reference contains C, the aligner reports a C in the read at that position. Otherwise, it reports a T.

Although there are some uncertainties involved, this is actually turns out to be a reliable method of aligning BS-seq data. The problem is that there is a fair amount of logic required to implement it, and it takes a lot of time to compute.

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

BS-seq read alignment is costly



- With the same tissue sample, BS-seq alignment is slower and less sensitive than non-BS-seq alignment
  - WGBS (whole genome bisulfite sequencing) read alignment is about 10 times slower than it is for WGS (whole genome sequencing)
  - A read aligner can find 2-3% more high-scoring mappings for WGS reads than for WGBS reads

- Workstation configuration
  - 12 CPU cores (24 threads) at 2.93GHz
  - Three Nvidia K20c GPUs
  - Arioc

Here are some data to show you how much harder it is to do BS-seq read alignments.

The numbers speak for themselves.

We obviously wanted to make things go faster by using a GPU – so let's take a look at how we attacked the problem.

[Data from a human hepatocellular carcinoma cultivar (https://www.ncbi.nlm.nih.gov/Traces/study/?acc=SRP117159). Thanks to the BGI for placing this data in the public domain!]

C, C_m, and T

- Convert all C to T and use a general-purpose read aligner
  - Use index structures where C is represented by T
  - Post-process to disambiguate T
  - Slow (even with a GPU-based aligner)
  - Potentially inaccurate
- Change the alignment algorithm to account for the C-versus-T ambiguity
  - C-to-T index structures are still needed
  - Faster (sequences remain in memory)
  - More accurate (alignment computation uses non-CT-converted sequences)
  - Amenable to GPU implementation

The general approach to doing BS-seq alignments is to treat all C as T, both in the reference and in the reads.

There are basically two ways to do this. One is to convert all C to T, do the alignments with a general-purpose read aligner that actually knows nothing about bisulfite-treated DNA, and then post-process the results to resolve the C-versus-T ambiguity. This kind of linear approach is tempting, but it has problems with both speed and accuracy.

This approach is slow for two big reasons:

- Alignments have to be emitted by the general-purpose aligner and then ingested by a second application that post-processes the reported alignments
- The aligner has to compute a lot of potential alignments for each read because there are many potentially-similar regions in the reference. For example, a region rich in C and T might contain only a small number of locations where the pattern of Cs and Ts in a read might match; if all the Cs are treated as Ts, the specificity of that pattern decreases.

There are also some potential inaccuracies:

- When C is represented by T, the read aligner may lack the information it needs to compute the best alignment, especially in regions rich in C and T.
- The specific case where a C in the read maps to a T in the reference has to be handled separately to avoid systematically failing to report Cm for such mappings. This is a costly speed-for-accuracy tradeoff.

The alternative is to embed the logic required to handle the C-versus-T ambiguity within the alignment algorithm. At first thought, this sounds awful, if for no other reason than that you can no longer treat the general-purpose read aligner as a "black box." But the advantages are clear:

- If alignments are computed directly between the raw, non-CT-converted read and reference sequences, that eliminates a couple of kinds of inaccuracies associated with aligning CT-converted sequences.
- The implementation involves less data movement, since the information required to disambiguate C from T is already memory-resident.
- And – most importantly – the same thread of execution can handle both the alignment computation and the C-versus-T logic. This means we can use GPU threads to do the work and hopefully get that 10x speedup that makes it worthwhile to use the GPU.

Algorithmic adaptations

- Mappings between C in the reference and T in the read are matches, not mismatches
  - This is crucial to making the algorithm report the best alignment
  - Alignment scores and mapping quality scores (MAPQ) do not need to be recomputed during post-processing
- Logic for computing reverse-complement alignments is implemented in CUDA threads instead of additional index structures

Let's look a bit more carefully at how we adapted the read-alignment algorithms for GPU acceleration.

The two key adaptations we made were chosen because it was possible to implement them on CUDA threads.

Probably the most important adaptation is obvious in hindsight: we changed the way alignment scores are computed. With non-bisulfite-treated reads, only matching symbols are scored with positive values. With bisulfite-treated reads, a T in a read that maps to a C in the reference is also assigned a positive value.

This simple change affects how alignment computations are implemented. In our software, we use two different algorithms to find mappings. One is the well-known Smith-Waterman algorithm, which can find mappings that contain gaps. Our Smith-Waterman implementation uses a little lookup table in which the scoring parameters reside, and it is straightforward to alter the contents of that table to score C (in the reference) with T (in the read) as a match.

It takes a bit more work to achieve the same result with the spaced-seed algorithm we use for non-gapped alignments. The implementation actually depends on the bitwise manipulation of binary-encoded symbols, so additional logic is needed to cause a T in a read to match a C in the reference. This surely slows the CUDA kernel down, but that doesn't affect the over speed of the application.

A less obvious adaptation involves handling reverse-complement mappings. DNA is double-stranded, but a DNA sequencer reports the sequence of only one strand per read. A BS-seq read aligner can deal with this by running a general-purpose read aligner twice, once with a C-to-T converted index and once with a G-to-A converted index, but it is much faster to handle reverse-complement mappings by computing the reverse complement of each read – again, in CUDA threads – and using that to probe an index or lookup-table structure.

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

C versus T in kernel code

```
static inline __device__ INT32 countMismatches( const UINT64 Rcurrent, const UINT64 Qcurrent, const bool baseConvCT, const UINT64 tailMask = _I64_MAX )
{
    UINT64 x = (Qcurrent ^ Rcurrent) & tailMask;        // each 3-bit symbol is nonzero where the symbols don't match
    UINT64 b = ((x >> 1) | x | (x << 1)) & MASK010;     // bit 1 of each 3-bit symbol is set where the symbols don't match

    /* For bsDNA alignments, C (101) in the references matches T (111) in the query:
        - in this situation, Q^R = 010, but this is ambiguous:
            R      Q
            C 101  T 111
            T 111  C 101
            A 100  G 110
            G 110  A 100

        - we disambiguate this from the three other possible ways that 010 can occur in the XOR string
            by zeroing bit 1 of each 3-bit symbol in the Q^R value wherever R contains C (101) and Q
            contains T (111)
    */
    if( baseConvCT )
    {
        UINT64 cr = ((Rcurrent >> 1) & (~Rcurrent) & (Rcurrent << 1));  // bit 1 of each 3-bit symbol is set where the symbol is C (101b)
        UINT64 tq = ((Qcurrent >> 1) & Qcurrent & (Qcurrent << 1));     // bit 1 of each 3-bit symbol is set where the symbol is T (111b)
        b &= ~((cr & tq) & MASK010);
    }

    // return the number of mismatches
    return __popc64( b );
}
```

For those of you who like code, here is a concrete example of what I've been talking about.

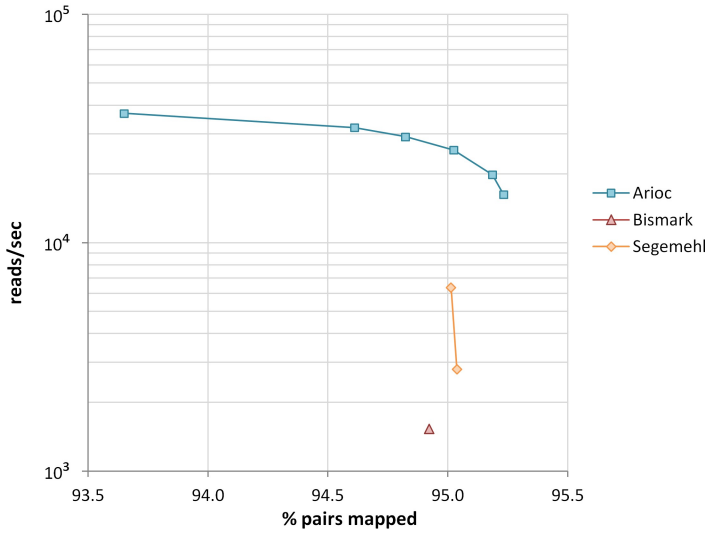
In the nongapped spaced-seed aligner, symbols are represented as 3-bit fields, and sequences are represented as 64-bit values that contain an ordered set of 3-bit fields. Counting mismatches between two sequences is thus carried out by doing an exclusive OR between two 64-bit values and counting the 1 bits.

When we're dealing with the C-versus-T ambiguity involved in bisulfite-treated read sequences, we need to do some additional work to cause a T in the read sequence to match a C in the reference. Although there's more logic here, there are no divergent execution paths and the extra computation involves only bitwise manipulation of values in GPU registers, so the impact on speed is negligible.

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

Speed versus sensitivity

- GPU-accelerated implementations
  - Arioc (2018)
- Multithreaded CPU implementations
  - Bismark (2011)
  - Segemehl (2012) – does not compute alignment scores, mapping quality scores, or methylation context



- Workstation configuration
  - 12 CPU cores (24 threads) at 2.93GHz
  - One Nvidia K20c GPU

The results are pretty good.

The basic result is that the GPU-accelerated implementation is at least 10x faster than comparable CPU-only implementations.

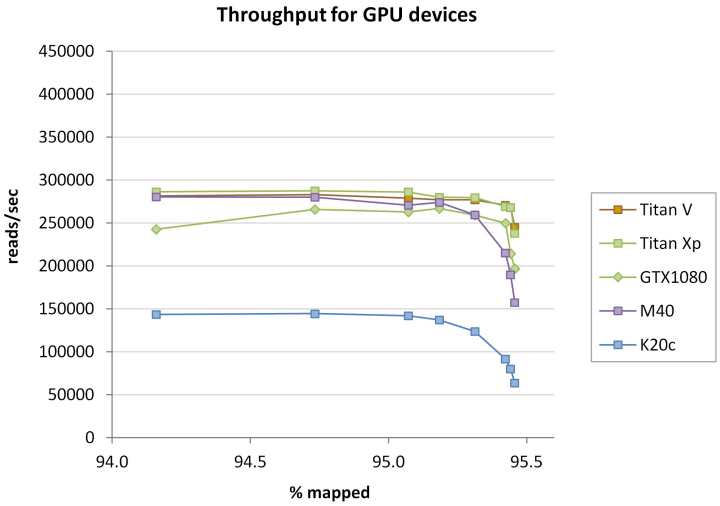
Other experiments show that it is just as accurate as the best of the CPU-only implementations.

The fastest CPU-only application isn't really comparable because it cuts some significant computational corners.

S8130: GPU-accelerated short-read aligner for bisulfite-treated DNA sequences

Speed and GPU microarchitecture

- GPU devices
  - K20c (Kepler)
  - M40 (Maxwell)
  - GTX 1080 (Pascal)
  - Titan Xp (Pascal)
  - Titan V (Volta)
- Why are newer microarchitectures faster?
  - More GPU threads in flight
  - Higher GPU memory bandwidth



- Workstation configuration
  - 12 CPU cores (24 threads) at 2.93GHz
  - One Nvidia GPU

This is GTC, so I have to show you what happens when you use newer GPU hardware.

Things get much faster, of course.

What's interesting is the big jump in speed from Kepler to Maxwell. The fact that we seem to hit a wall with newer GPUs is a hint that something besides raw compute speed is becoming important.

[The important hardware difference at that point may be that the K20 uses PCIe v2, and the newer GPUs are all PCIe v3.]

In fact, when we profile the application we can see alignments getting computed with increasing speed on the newer GPUs – for example, Volta computes alignments about 50% faster than Pascal.

We're not certain yet if the current bottleneck is about transferring data between the host system and the GPU, or if the CPU threads (which do string formatting and file I/O) are no longer keeping up with the data generated by the GPUs. All I can say right now is that we're working on it!

Lessons learned

- Sacrifice code simplicity to gain speed
- Modify algorithms so that they can be computed efficiently on GPU threads
- Evaluate performance carefully – try to get the same work done 10x faster
- A GPU is not a “magic bullet” – other factors can have a big impact on performance

To wrap things up...

We’ve just spent some time looking at concrete examples of how to make GPU acceleration work for a particular, focused problem.

Now I’d like to generalize a bit from our experience:

- GPU-based implementations are generally harder to build and maintain
- Speed comes from pushing computation onto GPU threads
- Be sure the performance gain is worth the effort

In this particular case, we’re happy with the results.

Now -- I’m open for questions and comments. Thank you again for coming to this session!

Questions / Comments

Arioc is available on Github: http://github.com/RWilton/Arioc

Our article about developing and testing Arioc with bisulfite-treated DNA appeared in *Bioinformatics*: http://doi.org/10.1093/bioinformatics/bty167