Computational Genomics

Accelerating the Variant Calling Pipeline

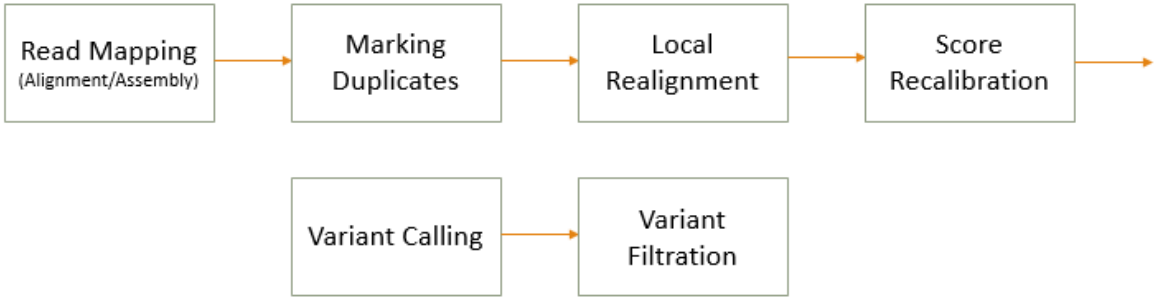
David Korobov
University of Illinois
May 13, 2016

Contents

1 Background

As genome sequencing continues to become less expensive, there exists a great potential for computationally-intensive solutions that assist the medical community in personalized diagnosis and treatment. The algorithms for implementing these solutions exist. However, the sheer magnitude of data that can be harnessed from sequencing technology leads to sluggish performance on regular CPUs. In improving the runtime of genomic analysis, we look at frequently occurring algorithms and bottlenecks that can be accelerated on GPUs and FPGAs. We hope to take advantage of computer architecture optimized for parallel computations and examine the tradeoffs involved in using these external devices.

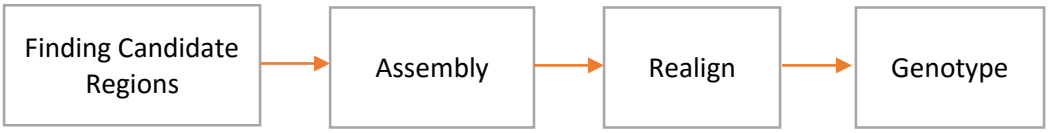
1.1 Variant Calling



A visualization of where variant calling fits into the genomic analysis pipeline.

My work this semester centered on implementing software that can be run on GPUs for the Variant Calling step. The Variant Calling Pipeline describes a mechanism by which

we can isolate the differences between a sample and reference genome. Once found, these genetic differences may be used in predicting predispositions to various diseases, including cancer. Variant Calling includes the following steps and algorithmic components:



Finding candidate regions: Entropy calculation, Gaussian low-pass filtering, thresholding
Assembly: De-Bruijn Graph Assembly
Realign: Pair-hidden Markov model
Genotype: Statistical modeling

In my work, I focused on the first two steps: finding candidate regions and De-Bruijn Graph Assembly.

2 Finding Candidate Regions

In this step, we determine what the candidate regions are for mutations in the reference genome. This step takes in three separate data structures from previous parts of the over-arching pipeline:

Nucleotide Array: This array consists of 32-bit elements. The first element is the number of short reads in the array, and the remaining elements hold short reads which are structured as follows: one element for the number of nucleotides in the read, followed by the data. Each nucleotide is represented as a 2-bit value.

# of short reads	short read #1	short read #...	short read #n

Cigar Array: This array is structured in the same way as the nucleotide array. However, the 2-bit nucleotides now represent either a match, mismatch, insertion, or deletion relative to the reference.

# of short reads	cigar string #1	cigar string #...	cigar string #n

Starting Array: This array consists of 32-bit elements. The first element is again the number of short reads, followed by the starting position in the reference of each corresponding short read in the nucleotide array. Note that the starting positions are in increasing order.

# of short reads	starting position #1	starting position #...	starting position #n

2.1 R-Tree and Frequency Table Generation

This first step is done on the CPU side and performs two separate functions. It takes in the data structures described above and creates a frequency array of the sample genome. For every position in the genome, we calculate the number of C, G, T, and A nucleotides and store these values in the corresponding location in the frequency array.

Num A
Num C
Num G
Num T

Frequency struct

While this is being created, we also build a one-dimensional R-Tree for quick indexing of short reads. Since we do not know the length of each short read, we have no way of knowing what the starting index of each one is in the nucleotide array. Using an R-Tree, we can insert each short read and its corresponding start and end positions in the reference. When we find candidate regions in the reference later on, we can search the R-Tree for the indexes of the short reads in the nucleotide array that belong to this candidate region.

2.2 Entropy Calculation

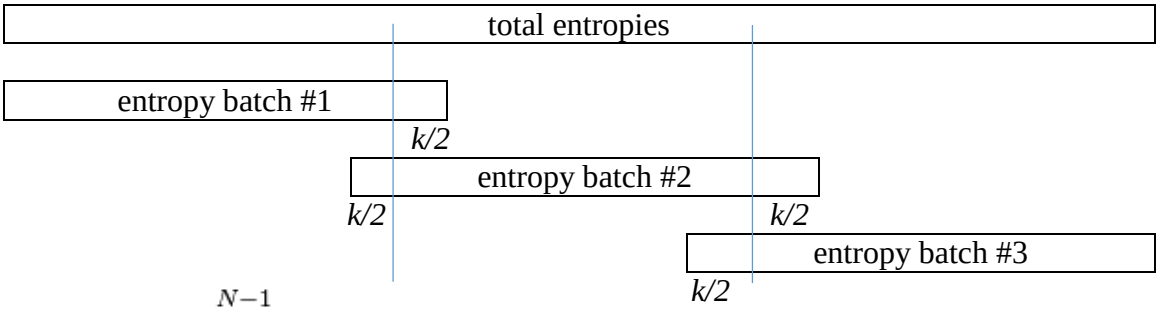
To discover possible areas of mutation in the sample genome, we calculate the Shannon entropy at each position of the frequency array:

$$H = -\sum_{i} p_i \log_b p_i$$

This is done via OpenCL and run on the GPU in parallel.

2.3 Guassian Low-Pass Filtering

Next, we take the entropies and perform a Gaussian convolution. This smooths out the calculated entropies to eliminate any anomalies in the sequencing. Note that convolution for a particular element requires k/2 elements before and k/2 elements after, where k is the filter length. Since the GPU can only handle about 4 GB of memory at a time, and the amount of data far exceeds 4 GB, it is important to pass in k/2 extra elements at the start and end of each batch of data for an accurate calculation of the convolution.



$$(f * g_N)[n] = \sum_{m=0}^{N-1} f[m]\ g_N[n-m]$$

This is done in parallel via OpenCL and run on the GPU.

2.4 Threshold Calculation

The threshold calculation is also run on the GPU and simply checks if the output of the convolution is greater than some threshold. If greater, the output of the threshold kernel is a Boolean one. This signifies a problem area in the genome that may have a mutation.

2.5 Creating De-Bruijn Graph Assembly Input

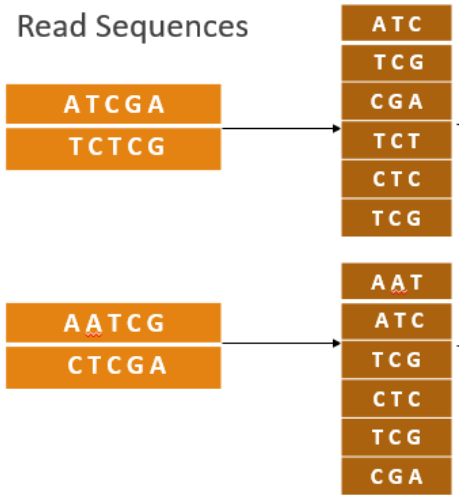
The Boolean array from the threshold kernel is now used to find short reads that overlap the potentially mutated regions. To do this, we traverse the threshold array and find the start and end index of any sequences of 1's. This sequence indicates a problem region. We then search the one-dimensional R-Tree to find the start and end indexes of any overlapping short reads in the nucleotide array. Once found, we copy these short reads into a single array and use it as an input to the De-Bruijn Graph Assembly.

3 De-Bruijn Graph Assembly

De-bruijn graph assembly is used to analyze the candidate regions and determine which differences are due to sequencing error and which are due to mutation. I implemented the first half of this algorithm on the GPU – the remainder was done by Safa on the CPU.

3.1 K-mer Generation

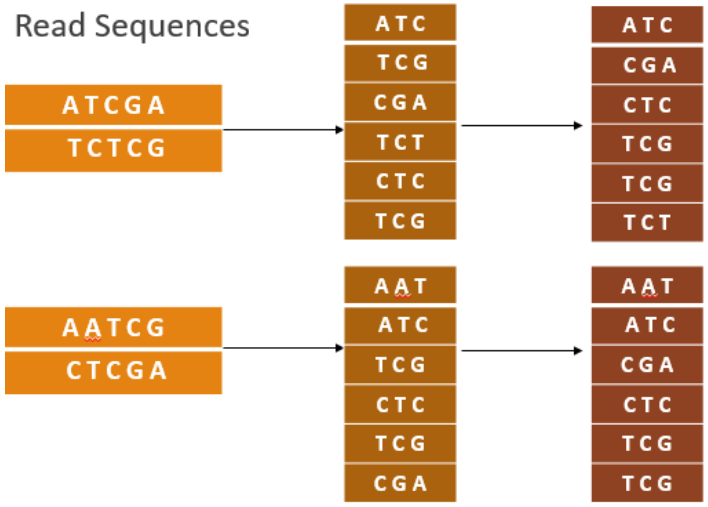
The first step of the assembly consists of splitting up the nucleotide data from the short reads into k-mers. K-mers are simply subsequences of the short read of length k. For instance:



As before, due to limitations in GPU memory, we process the short reads in batches.

3.2 Bitonic Sort

Once we have created all the k-mers, we take advantage of the parallel architecture and perform a bitonic sort on the GPU. Since a bitonic sort requires a power of two input, we add the necessary padding. The code for this was provided by Intel.



3.3 Frequency Reduction

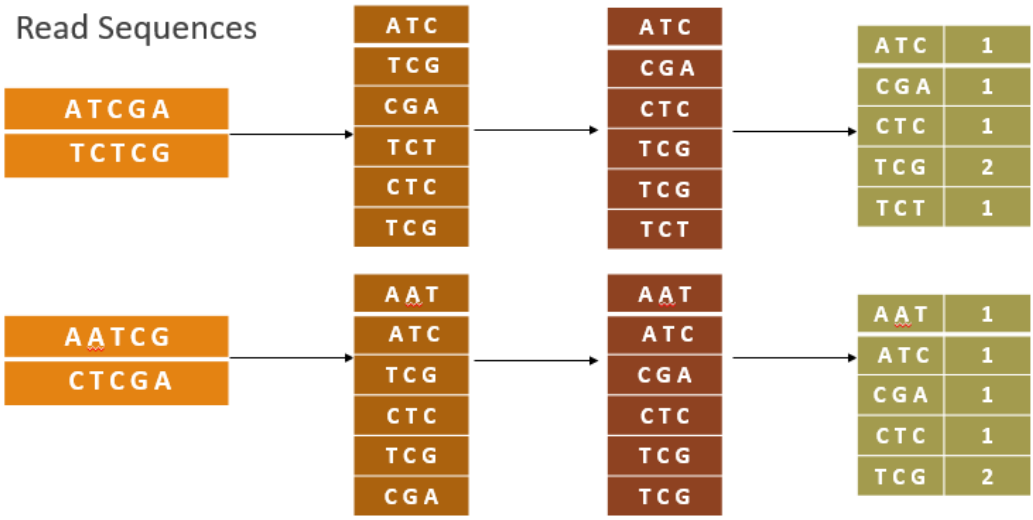
The goal here is to create a giant table of k-mers and their associated frequencies. We encountered two main problems in this step and the subsequent reduction.

1. We wanted to synchronize the threads to ensure proper execution, using barriers in the kernel implementation. However, the GPU can only process a limited number of workgroups at a time. This meant that if we enqueued too much data to the GPU at once, only a portion of the workgroups would execute. The GPU would wait until all the workgroups arrived at the synchronization barrier, but the non-executing workgroups would wait until the others would finish. This caused a deadlock.

2. It was not obvious how to create a single frequency table using parallel programming. A CPU implementation is simple – you traverse the sorted array, count the frequency of each element, and place this value in the output array. A parallel implementation is much less intuitive, since a) the starting location of each thread may not be at the start of a new k-mer and b) it is impossible to know where the frequency of each k-mer, once calculated, should be placed in the output array.

Finally, we found an implementation that works. We pass in a fixed chunk of data to the GPU at a time to avoid deadlock. We perform the reduction by initially comparing every two elements to each other. If they match, we increment the frequency count for that particular k-mer and take note of the start and end index of these miniature frequency tables. Then, we compare frequency tables of a maximum size two, then four, and so on,

incrementing the k-mers that appear multiple times. In the end, we successfully created frequency tables that were used in the graph, done by Safa.



4 Considerations

There were a number of considerations as I was writing software in OpenCL.

1. We wanted to limit data transfer as much as possible. Copying data between the CPU and GPU is expensive and can negate any performance advantages we get from the parallel architecture.

2. Barriers limit the amount of data we can pass into the GPU at a time and worsen performance. Unless strictly necessary, do not synchronize threads.

3. Determine the optimal grain (work-per-thread) size. OpenCL allows for flexibility when determining the size of each work group and the amount of work done by each thread. Optimal values are impossible to know without thorough testing.

4. Local testing does not guarantee the correct execution on the Compgen cluster. For some reason, my laptop did not catch certain bugs in my code. Once I tried remote testing at IGB, my program crashed.

5. Limit the number of conditional statements as much as possible. GPU performance falters at conditional statements – the architecture is not designed for it. Replace it with mathematical operations that mimic its behavior.

6. Conserve as much memory as possible on the GPU. Since the GPU can only hold 4 GB of memory at a time, it's crucial to free any buffers that we are no longer using.

7. Optimize code. This is the first project I've worked on where speed is of utmost importance. On the GPU size, we replaced a struct of four uint32_t types with an int4. This takes advantage of the SIMD unit which performs computations using these struct variables in parallel.

5 Conclusion

This was my first semester doing research, but I am very pleased with the experience and what I was able to accomplish. Never having taken parallel programming before, it was a brand new introduction to OpenCL and all the related terminology: barriers, workgroups, context queues, types, and so on. I was also poorly versed in the biological component to the project, and spent a fair amount of time researching the makeup of DNA and the type of anomalies we may be looking for. That said, Subho was very helpful and I got to work quickly. Safa's contribution was also invaluable. Together, we were able to complete two main functions in the Variant Calling Pipeline and test them on the Compgen cluster.

Next semester, I intend to continue accelerating the algorithms involved in Variant Calling on the GPU and put the software already developed on FPGA.