GPU Multiple Sequence Alignment
Fourier-Space Cross-Correlation Alignment

Final Project Writeup
EN.600.639 Prof. Ben Langmead
–

Matthias Alexander Lee

May 3, 2013

Abstract

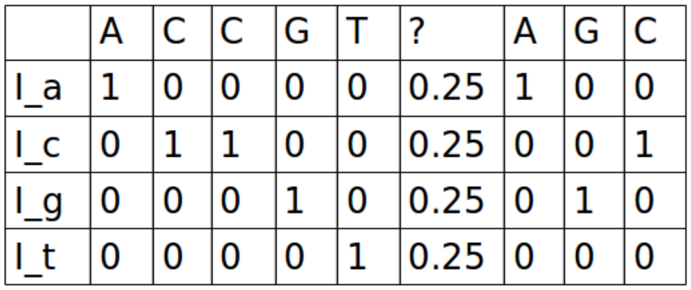
The aim of this project is to explore the possible application of Graphics Processors (GPUs) to accelerate and speed up sequence alignment by Fourier-space cross-correlation. Aligning signals using cross-correlations is a well studied approach in the world of signal processing, but has found relatively little reception in the realm of computational genomics. As long as we can treat DNA as a signal by encoding it numerically, we can utilize these cross-correlations to align and compare 2 or more strands of DNA or RNA. Fourier-space cross-correlations have a favorable computational complexity of $O(n \log_2(n))$, where n is the length of the longer input strand. A single cross-correlation consists of three FFTs and a sliding dot-product, both of these types of operations are inherently parallel. Due to the extraordinary length of DNA sequences and the independence between operations, we can extort parallelism to a high degree. Therefore this problem maps very well to the highly parallel architecture of the modern GPU. This project explores the method, execution and performance of GPU-based DNA/RNA alignment using cross-correlations.

Introduction

Much of genomics depends on some kind of alignment, whether multiple sequence alignment or alignment of sequence reads to a reference genome. Alignment is a very important, but also a computationally complex problem. Approaches such as Needleman-Wunsch or Smith-Waterman rely on dynamic programming while others use Suffix trees or the Burrows Wheeler Transform. An interesting and little studied approach is alignment by cross-correlation. Alignment by cross-correlation yields both regular base-by-base matches as well as complementary matches indicating matching nucleotides from the complementary DNA strand. Outside of the field of genomics, cross-correlation is a well studied method, especially in the field of signal processing where it is often used to determine similarities and phase differences between signals. These signals are usually in the form of time series, though in our case we will ignore the implied relation to time and just consider it a sequence of intervals.

Prior Work & Motivation

The idea of using Fourier-space cross-correlation was first entertained by Joseph Felsenstein[3] et al. in 1981. Felsenstein describes a technique which creates four separate indicator arrays of binary values, one for each nucleotide. Each of these arrays is used to encode the occurrence and location of each different nucleotide by marking the location with a 1.0. For example, lets define four indicator arrays, $\{I_a, I_c, I_g, I_t\}$, *see Figure 1.*

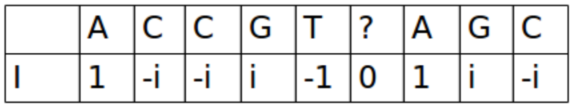


	A	C	C	G	T	?	A	G	C
I_a	1	0	0	0	0	0.25	1	0	0
I_c	0	1	1	0	0	0.25	0	0	1
I_g	0	0	0	1	0	0.25	0	1	0
I_t	0	0	0	0	1	0.25	0	0	0

Figure 1: Translation between DNA and numeric values

When a nucleotide at a position is unknown we simply distribute our count of 1.0 evenly across all indicator arrays at that position(see position marked with a ?). This method can be extended to also reflect more complicated uncertainty values such as $\{0.6, 0.1, 0.1, 0.2\}$, $\{I_a, I_c, I_g, I_t\}$ respectively. The initial approach outlined by Felsenstein aligns two input sequences, X and Y, by first transcribing them each into their indicator arrays and then using four time-domain cross-correlations to find the best alignment between the two sequences. Due to the less-than-favorable $O(n^2)$-complexity, Felsenstein then suggests using a frequency-domain FFT-based cross-correlation to vastly decrease the complexity to $O(n \log_2(n))$. *See section below for details on time-domain and frequency-domain cross-correlations.* Even though Felsenstein's second approach was good at reducing the $O(n^2)$-complexity, it still required one cross-correlation for every indicator array. Both Cheever[1] et al. and Rockwood[4] et al. improved on Felsenstein's method by suggesting the use of only one indicator array. Instead of transcribing into multiple arrays of binary values we now transcribe into a single array of complex values, A,T,C,G transcribes to 1,-1,i,-i. (*see Figure 2*) This reduces the computational effort even more by eliminating multiple FFTs. Rockwood also explored a method of graphing the partial sum of the real parts of the cross-correlation's result to visualize where the similarity between two inputs lies.

There are few mentions, besides the above, of this method through out the genomic literature. Much of the computational challenges associated with sequence alignment have focused on aligning short sequences to much larger sequences. Since a cross-correlation needs to perform Fourier-transforms of the size of the longer sequence, it cannot contend



	A	C	C	G	T	?	A	G	C
I	1	-i	-i	i	-1	0	1	i	-i

Figure 2: Translation between DNA and single series of complex values

with other methods available for short sequence alignment. But as the shorter sequence grows closer to the length of the larger, it becomes more efficient. Due to the inner workings of an FFT, the FFT-based cross-correlation is most efficiently when both input arrays are of the same size and when that size falls on a power of two.

Since the algorithm mostly consists of FFTs and element-wise multiplications, it is an ideal candidate for parallelization. Especially because the larger the input arrays become, the more inherent data-parallelism is present. For this reason Graphics Processing Units (GPUs), with thousands of cores, provide a perfect platform for accelerating these calculations.

Cross-Correlation

A time-domain cross-correlation is nothing more than calculating the sum of the element-wise products between two inputs of the same length. Lets call our two input arrays of length n, h and g, and lets call this sum f. f indicates the similarity/correlation between h and g at their default alignment.(*see Figure 3*)

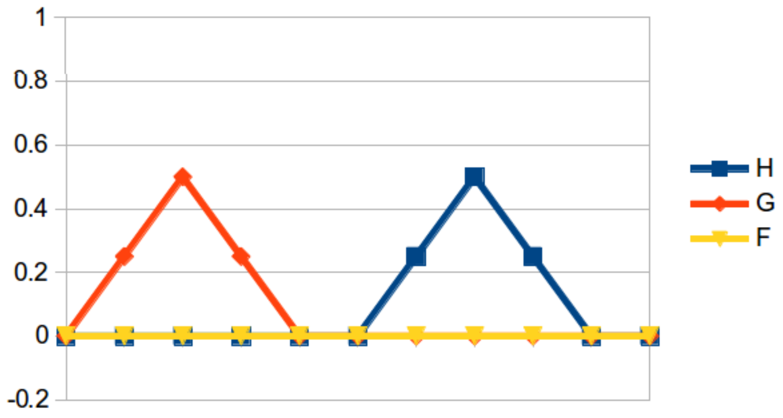


Figure 3: Sequence h and h and their element-wise product f

$$f(k) = \sum_{j=0}^{n} (h(j) \cdot g(j+k)) \tag{1}$$

To find their correlation at different alignment, we start shifting g with respect to h so that the overflow at the end of g wraps around to the beginning. Lets call the value of this "shift" k and the sum at that "shift"-position $f(k)$. Now if we compute this sum for every possible k, then locate the maximum $f(k)$, we have found the shift yielding the best alignment. (*see Equation 1 and Figure 4*) This approach provides a straight forward alignment indicating which shift maximizes the overlapped between our two sequences. Sadly, this approach grows at the unfavorable complexity of $O(n^2)$. Luckily

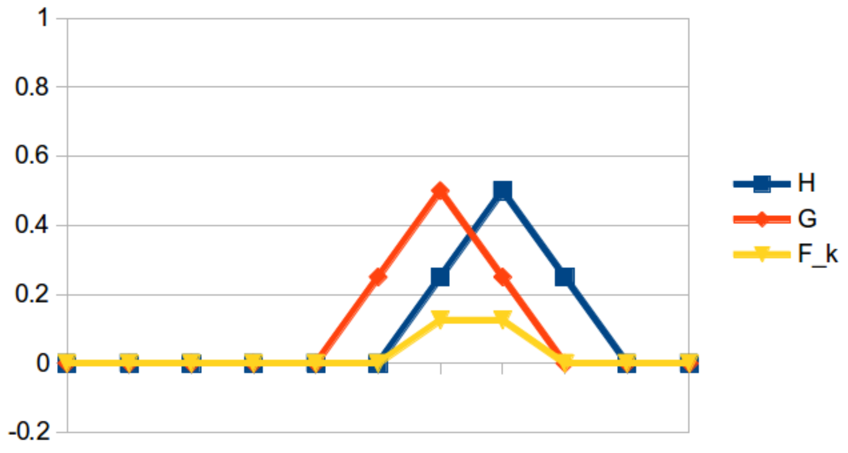


Figure 4: Sequence G shifted by 4 points, k=4

there is an equivalent algorithm which achieves $O(n \log_2(n))$-complexity, the FFT-based Fourier-space cross-correlation. The FFT-based cross-correlation achieves it's boost in efficiency by using the Cooley/Tukey[2] Fast-Fourier-Transform. This second algorithm transforms it's two input sequences, h and g, of length n from the time-domain into the frequency-domain. Let the $FFT(h) = H$ and $FFT(g) = G$. Now if we calculate F, the element-wise product of H and the complex conjugate of G. Then when we transform F back into the time-domain, we end up with an array of correlations, call it f. Each element in f corresponds to the correlation at a shift equal to that element's index. For example, if $f(6) = 0.5$, the correlation between h and g, when g is shifted by 6 elements, the correlation is 0.5. If we simply find the maximum and its corresponding index in f, we have the shift producing the best correlation and therefore the best alignment. (*see Equations 2, 3 and 4*)

$$F = FFT(h) \cdot \texttt{conj}(FFT(g)) \tag{2}$$

$$\texttt{best correlation} = \max(iFFT(F)) \tag{3}$$

$$\texttt{best shift} = \texttt{arg}\max(iFFT(F)) \tag{4}$$

Implementation

The tool written for this project is implemented in python, with dependencies on NumPy, pyCUDA and pyFFT.CUDA. At a most basic level, there are 3 main parts: pyFFTalign.py, dataObj.py and aligner.py. Each of these breaks out the 3 main areas of code, pyFFTalign.py is a wrapper which allows for command-line arguments and also controls the reading in of Sequence data. dataObj.py is an object containing the sequence data itself and the methods for modifying that data. To instantiate a dataObj we pass it the name of the sequence, the source filename and the raw DNA or RNA reads. Upon instantiation, it automatically transcribes the raw DNA/RNA into a complex indicator array. There is also functionality allowing for verification of successful transcription, as well as returning of the raw and transcribed array padded to a specified size. aligner.py is where all the magic happens. There are two main functions, the CPU-implementation is calcCorrShiftmn() and the GPU-implementation is in calcCorrShiftGPU(). Both functions are very similar as the use of pyCUDA.GPUArray allows very similar functionality as NumPy.ndarray. The code mostly differs due to the setup necessary for the CUDA accelerated FFTs.

For simplicity we will discuss the CPU implementation and then contrast the changes necessary for the GPU acceleration. calcCorrShiftmn() takes in two main parameters, each being an array of dataObjs, $H[\ ]$ and $G[\ ]$. We begin by looping over every H contained in $H[\,]$ and for every H we will iterate over every G in $G[\,]$. This will give us every combinations of H and G. For every pair of H and G, we calculate a FFT-based cross-correlation. Since the FFT algorithm is most efficient when the input size is a power of 2, we must ensure both our inputs are zero-padded to the next power of 2. We then execute compute the FFT of each of them, H_{-} and G_{-}, followed by an element-wise product between H_{-} and the complex-conjugate of G_{-}, the resulting array is immediately transformed back out of Fourier space by way of an inverse-FFT. This resulting array, f, now contains the correlations for every possible offset, next we locate the maximum value in f and it's corresponding index. These are the best correlation value and the offset at which it occurred. (*see Figure 5*)

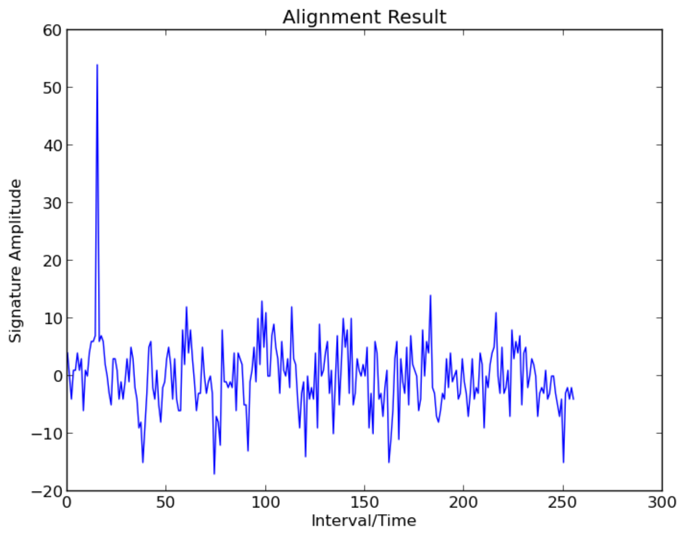


Figure 5: Array of correlations, note the strong peak indicating a good alignment at around 19.

The GPU equivalent of the above function only varies slightly. The looping, padding and math is identical, the main difference is that we have to start by loading our arrays into GPU memory. We do this by making use of pyCUDA's GPUArray, which supports similar arithmetic and vector methods as the NumPy's ndArray. Most importantly GPUArray also supports element-wire broadcasting of operations, so we can just call $Z = X * Y$ and we get an array of element-wise products of every element in X and Y. The next difference is the FFT "plan" setup. An FFT "plan" is precomputed and precompiled configuration of an FFT of a certain size and the parameters that go along with it. The code currently has "wait_for_finish" set to true so that the code doesn't advance ahead while the GPU is still calculating. The rest of the GPU implementation is mostly the same, with the exception of needing to return our resultant array back to CPU memory before we identify the best correlation.

Now that we have discussed the procedure of how we arrive to our result array f lets have a look and see what information we can gleam out of the results besides what shift provides the best correlation. In the introduction we also mentioned that this method of alignment can give us complementary matches, *see Figure 6* for match resulting from the complement of the sequence used in Figure 5. Both of these figures match exactly, but often in genomics we have partial matches. When we have a sequence with read errors, specifically replacements, we will still see a peak at the same locations, but the correlation will be lower. If we have errors such as insertions or deletions, we will get "split peaks", signifying we have matched 2 pieces, but at different shifts. *see Figure 7 for example*

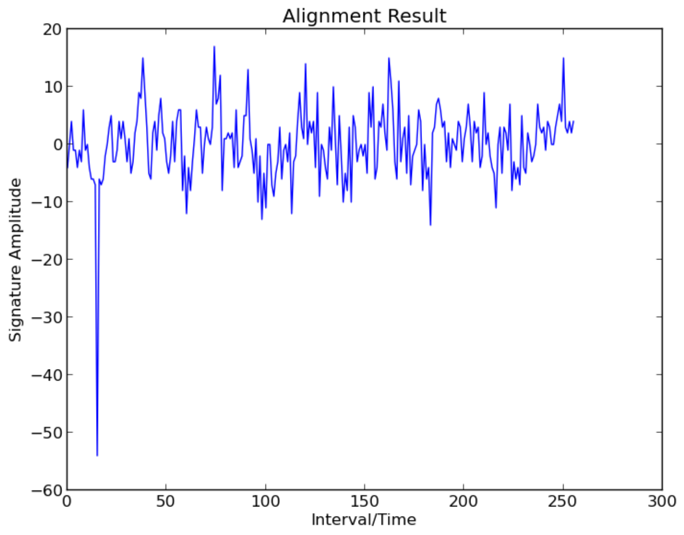


Figure 6: Array of correlations, note the strong negative peak indicating a good complementary alignment.

Results & Conclusion

The original goal of this project was to compare the possible performance improvement gained by utilizing a GPU instead of the CPU. The results look quite good, the GPU had

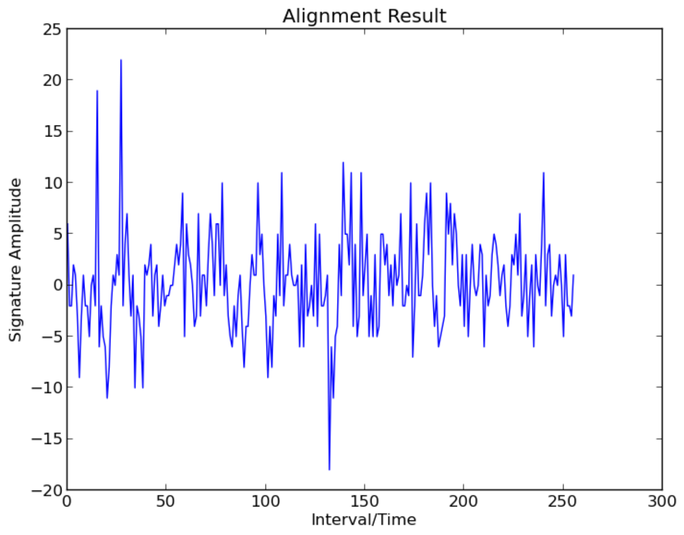


Figure 7: Array of correlations, "Split peaks" occur with insertions or deletions are matched.

an over 10x speedup in comparison to the CPU implementation. (*see Figure 8*) There is

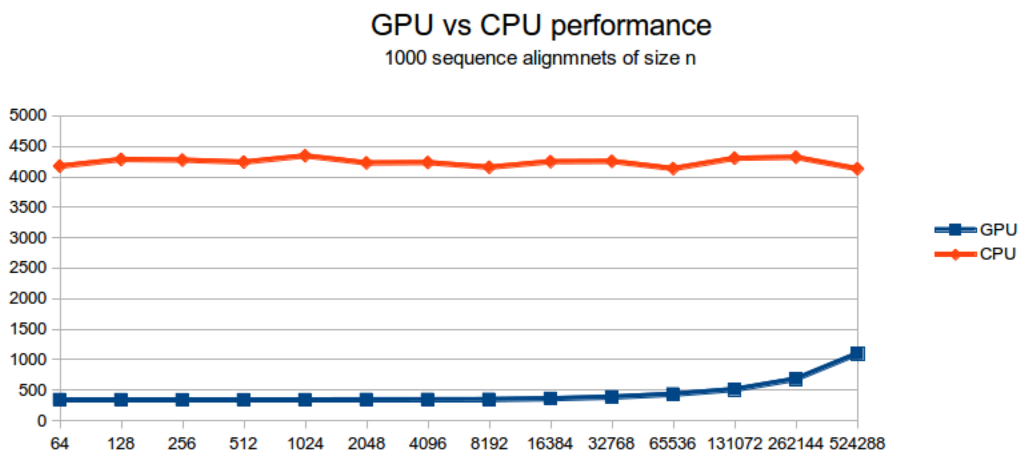


Figure 8: performance of matching 1000 sequences of length n against a sequence of length 5.3m

an interesting increase in the computation time as n becomes larger. Some of this was due to the zero padding that had to be done in order to align both inputs to a power of two. At first the code would just round up to the next power of two and will the rest with zeros. After optimization the code to break the array down into two smaller input sets, this artifact went away. (*see Figure 9*)

The current GPU implementation has a lot of room for improvement. First of all cuFFT, the CUDA FFT library provided by NVIDIA, runs a slightly faster than pyFFT.CUDA. Second currently the implementation waits on each FFT to finish before

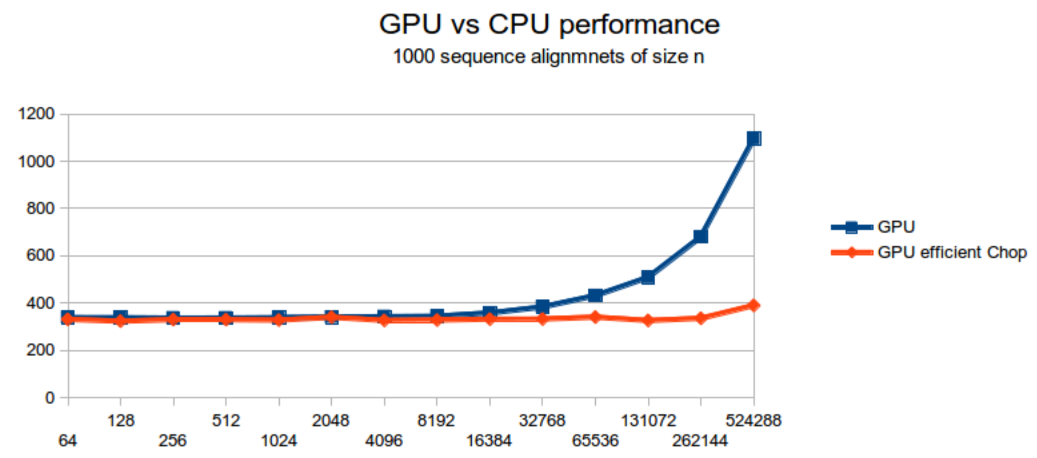


Figure 9: performance of matching 1000 sequences of length n, GPU code vs optimized working set splitting

it proceeds. This could be improved, by using streams to line up work on the GPU, ensuring it does not get to sit idle while the CPU continues processing and readying more data. Running just one stream of this GPU code only partially utilized the GPU's full potential. To processing even more, multiple streams could be utilized to schedule and execute multiple FFTs and element-wise multiplications in parallel. This all would very much complicate the code and is out of the scope of this project, but for future work this could be achieved.

The results of this code quite impressive, but as mentioned above we could go much further with some more optimization.

References

[1] EA Cheever, DB Searls, W Karunaratne, and GC Overton. Using signal processing techniques for dna sequence comparison. In *Bioengineering Conference, 1989., Proceedings of the 1989 Fifteenth Annual Northeast*, pages 173–174. IEEE, 1989.

[2] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[3] Joseph Felsenstein, Stanley Sawyer, and Rochelle Kochin. An efficient method for matching nucleic acid sequences. *Nucleic Acids Research*, 10(1):133–139, 1982.

[4] Alan L Rockwood, David K Crockett, James R Oliphant, and Kojo SJ Elenitoba-Johnson. Sequence alignment by cross-correlation. *Journal of biomolecular techniques: JBT*, 16(4):453, 2005.