A GPU Algorithm for Comparing Nucleotide Histograms

Adrienne Breland[†] Harpreet Singh[†] Omid Tutakhil[†] Mike Needham[†]
Dickson Luong[†] Grant Hennig[‡] Roger Hoang[†] Torbjorn Loken[†]
Sergiu M. Dascalu[†] Frederick C. Harris, Jr.[†]

[†]Department of Computer Science & Engineering [‡]Department of Physiology & Cell Biology
University of Nevada, Reno
Reno, NV 89512

Abstract

We designed and implemented an algorithm for conducting bioinformatic sequence comparisons with GPU processing. The input is a set of short "test" sequences and a set of longer "reference" sequences. The algorithm determines where along any of the reference sequences the test sequences may align. We compared run-times of the GPU implementation with a CPU implementation on a large, real dataset and gained a 5.5-6x speedup.

1 Introduction

Utilizing the massively parallel architectures in GPU computing has led to exceptional increases in speed for many sequence comparative applications in bioinformatics [1]. However, this requires that algorithms be redesigned to exploit GPU hardware [2, 3]. For example, GPU adaptations of BLASTP, Smith-Waterman [4, 5, 6], and HMMER [7] have allowed speedups of 10x, 10x-50x, and 60x-100x respectively. As the advantage of GPU computing becomes more apparent, several applications are being designed to directly exploit GPU architecture while bypassing most computation on the CPU. These include applications such as Cmatch for sequence matching [8] which enables a 35x speedup, and MSA-CUDA for multiple sequence alignment [9] which also allows up to 37x speedup.

GPU processing is the execution of computation using a Graphics Processing Unit (GPU), stored on graphic cards, instead of a Central Processing Unit (CPU). A GPU may be thought of as a CPU which has been redesigned for a specific type of problem. CPUs have been optimized to execute sequences of instructions rapidly. These instruction sequences do not always remain the same, and often take different paths depending on execution at runtime. This is referred to as branching. Much of the CPU architecture is allocated to handle branching while still allowing rapid execution of instructions [10]. However, more parallel applications, such as graphics processing, are not characterized by high levels of branching. The same set of instructions are applied to large sets of data, such as making the same change to each pixel in an image. In these cases, CPU resources allocated to optimize instruction branching are not being used as efficiently as they could be. GPUs are not optimized for instruction branching and the majority of resources are left for direct computation [10].

Many algorithms used for sequence comparisons are well suited for implementation on GPU processors. Sequence comparisons are often embarrassingly parallel, meaning that comparing one pair of sequences may be accomplished independently of a simultaneous comparisons of another pair of sequences. Thus, the same set of comparison instructions can be applied to different sequence pairs and run in parallel. This type of parallel computation allows a well defined set of instructions with little to no branching.

Here, we present an algorithm for comparing sets of short RNA sequences against sets of reference sequences in which the majority of computation is accomplished on the GPU. The algorithm is used to identify sub-strings in the reference sequences with the potential for high alignment scores with any of the reference sequences. These are candidate regions for more computationally intensive, full alignment comparisons in later processing steps not described here.

2 Overview

The sequence data that we consider here consists of a set of short RNA sequences, which we call test sequences, and a set of longer reference sequences. Our test set contains 61,120 sequences which range in length from 15 -169 nucleotides. Our reference set contains 608 sequences which range in length from 21-700,395 nucleotides. The goal is tofind at what positions, if any, each test sequence may align somewhere along one or more of the reference sequences.

This is accomplished by computing and comparing the nucleotide histogram of each test sequence with the histogram of each subsequence (equal in length to that of the test) found in the references. If the nucleotide histograms are equivalent by a given percentage, which may be set by the user, the test sequence and the reference subsequence are marked as potential matches.

2.1 Complexity

Given N test sequences and M reference sequences, NxM string comparisons are accomplished. To perform a comparison between a single test sequence $(T_i, i \in N)$ and a single reference sequence $(R_j, j \in M)$, T_i must be compared with every overlapping subsequence in R_j with a length equal to that of T_i, denoted as $|T_i|$. This yields $(|R_j| - |T_i| + 1)$ comparison operations for each test sequence.

Assuming that each R_j is at least as long as the shortest T_i, which we call $|T_{MIN}|$, the maximum possible number of comparison calls will be:

$$Nx(\sum_{j}^{M}(|R_j| - T_{MIN} + 1))$$

$$N\sum_{j}^{M}(|R_j| - NMT_{MIN} + NM$$

assuming that:

$$\sum_{j}^{M}|R_j| >> MT_{MIN} \tag{1}$$

and:

$$\sum_{j}^{M}|R_j| >> M \tag{2}$$

we have:

$$O(N) = Nx\sum_{j}^{M}|R_j| \tag{3}$$

2.2 Histograms

The nucleotide histogram of a genomic sequence is an integer array of length four containing counts of each character type. The alphabet for any genomic sequence contains four characters $\{a, c, g, t/u\}$. DNA sequences contain 't' where

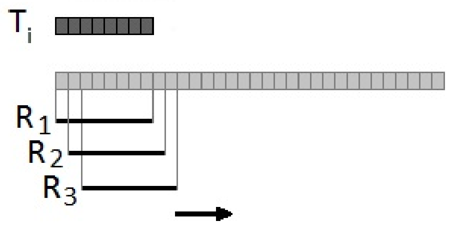

Figure 1: For each subsequence of length $|T_i|$ in the reference, (R_j), we compute a nucleotide histogram to compare with that of T_i.

RNA sequences contain 'u', however we may consider them equivalent when performing a string comparison, and in the following discussion, we only refer to 't'. Thus, for each test sequence T_i of length $|T_i|$, we compute the histogram in which the first element contains the number of a's found in T_i, the second element the number of c's, the third element the number of g's and the fourth the number of t's . Histograms are computed for each test sequence, and then again for each overlapping subsequence in the reference. This comparison scheme is illustrated in Figure 1.

Histogram differences are computed as the summed ratio of differences computed between counts of each nucleotide. Thus for two arrays (T_i and R_j) of length four, the difference in nucleotide counts (D) is given as:

$$D_{i,j} = \sum_{x=0}^{3} abs(T_i[x] - R_j[x]) \tag{4}$$

The percentage similarity $(S_{i,j})$ is computed as unity minus the percentage in difference between the two sequences:

$$S_{i,j} = 1 - \frac{D_{i,j}}{|T_i|} \tag{5}$$

Thus, if $S_{i,j}$ is greater than or equal to a user defined threshold, that region in R_j is flagged for further comparison with T_i.

3 GPU Implementation

The implementation language was Compute Unified Device Architecture (CUDA) C, a development language that enables efficient usage of the GPU architecture. The Thrust library was also utilized. This library contains many data structures and functions applicable to string processing which have been optimized for GPU architecture. Programs were developed and tested using a Fermi Architecture NVIDIA graphics card, GeForce 480 series.

3.1 Inclusive Prefix Sum (Scan)

Nucleotide counts are computed with the scan algorithm [11], which is implemented in parallel in the Thrust library for CUDA C. The scan algorithm sequentially compounds a binary operator function on an array of values, producing a new array of the compounded value up to each index. We use the *inclusive* version of scan, which requires a minor modification to input arrays because the Thrust function is *exclusive*. With the inclusive scan operation, the output at each index includes the compounded operation on the input at the same index. The exclusive scan only includes the compounded operation on the input up to the preceding index.

An example of the inclusive scan is given. Assume the binary operation is "+" with an input array of:

$X = [1, 1, 2, 3]$

The resulting output array will be:

$$X' = [(1), (1+1), (1+1+2), (1+1+2+3)]$$
$$X' = [1, 2, 4, 7]$$

To compute nucleotide counts in a sequence, we first divide it into four binary representations denoting the presence or absence of each nucleotide. Given a nucleotide sequence S, it is first converted to four binary strings (S_a, S_c, S_g, S_t) representing the presence or absence of nucleotide $x, x \in \{a, c, g, t\}$. This is illustrated in Figure 2. The binary conversions are accomplished using the thrust::transform function.

The inclusive scan algorithm is run on each of sequence permutation (S_a, S_c, S_g, S_t). This enables a parallel computation of the number of each character type in a given string or subsequence of a string. For example, assume a nucleotide string:

$S = attggaaaacacaa$

for which:

$S_a = 10000111101011$

An inclusive scan with the "+" operator results in:

$S_a' = 11111234556678$

The number of a's found in any contiguous portion of S may be computed by looking up only two indices S_a'. For example, to compute the number of

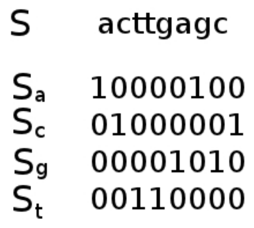

Figure 2: Binary representations of each nucleotide type.

a's in the substring beginning at the second character position and ending at the tenth ($S[1]$ to $S[9]$), we compute $S_a'[9] - S_a'[0] = 5 - 1 = 4$. This is illustrated in Figure 3.

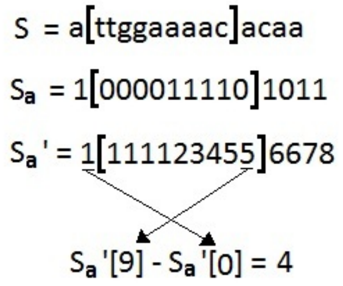

Figure 3: Computing the nucleotide count for a given substring only requires accessing two array indices.

4 Algorithm

4.1 Processing Test Sequences

The algorithm begins by concatenating all short RNA test sequences into one string with character delimiters inserted between each test sequence. This concatenation is converted into the four character specific binary strings, with delimiters maintained, as shown in Figure 2. An inclusive scan is then performed on each string. The scan compounds the "+" operator up to each delimiter, and then restarts. Thus, each numeric value in the scan output preceding a delimiter represents the number of characters of given type in a test sequence. The thrust::gather function is used to extract these values which precede delimiters thereby producing a condensed array with character counts per sequence. This is illustrated in Figure 4.

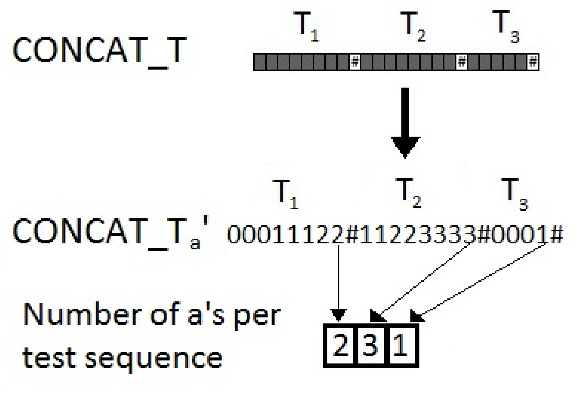

Figure 4: The thrust::gather function is used to extract nucleotide counts per sequence and condense them into an array.

4.2 Processing Reference Sequences

Reference sequences are also converted to four character specific binary strings and processed with the inclusive scan. Recall, to compare a test sequence (T_i) against a reference sequence (R_j), the character histogram of each overlapping subsequence of length $|T_i|$ in R_j must be computed (Figure 1). We assume that the test sequences are sorted by length, shortest to longest, prior to being used as input. Thus, for each length found in the set of tests, overlapping reference histograms are computed using the operation described in Figure 3 and results are stored temporarily. If several test sequences with the same length exist, the reference histograms for that length must only be computed once for comparison with all tests of that particular length.

5 Streaming

Streams are sets of instructions to be executed sequentially. Multiple streams can enable computational speedups by overlapping data transfer operations (between host (CPU) and device (GPU)) and data processing on the GPU. Ideally, the data transfer step in one stream will occur simultaneously with processing in another stream.

To make use of streams in our algorithm, test sequences are compared with all reference sequences and results are copied back to the CPU. Stream are managed by CPU threads which serve as a small process managers by calling a stream to compute comparison results for a test sequence on the GPU, and then transfer those results asynchronously back to CPU. When the stream has finished, the thread uses the next available stream to process the next available test sequence. Finding the optimal number of concurrent streams was accomplished by timing the process with stream number ranging between 1-100. The optimal number for our data set was 22. Figure 5 shows the graph of computing time vs. number of streams.

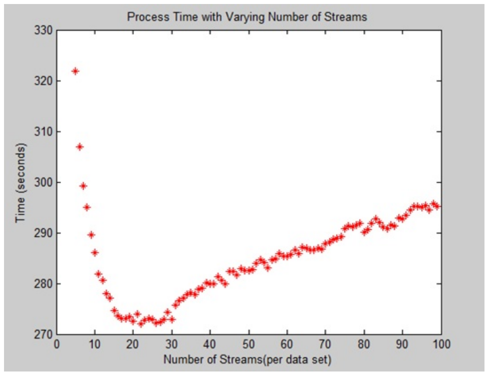

Figure 5: Compute time vs. number of streams.

5.1 Pseudocode

The pseudocode is provided in the following. In the pseudocode, R is the set of reference sequences and T_i represent a test sequence in the set of N input test sequence where $i \in N$. It is also assumed that the input sequences are sorted by length and that lengths are precomputed.

```
start compare()
    copy CPU → GPU (R)
    concatenate($T_i ... T_N$)
    copy CPU → GPU (CONCAT_T)
    scan (CONCAT_T)
    while $i < NUM$
        with next available stream
            $L = |next\ available\,(T_i)|$
                get $Histogram_L\,(R)$
                $\forall j, j \in M$
                    compare($Histogram\,(T_k)$,
                    $Histogram_L\,(R_j)$ )
            store (results)
        end stream
    copy GPU → CPU(results)
end compare()
```

6 Run-Time

We compared the GPU application run-time with a CPU based implementation. Both a GPU and CPU versions were tested on an Intel Core 2 Quad with 9GB of RAM running at 2.83GHz.

Like the GPU algorithm, the CPU version also compares nucleotide histograms of all test sequences against histograms of all overlapping subsequences in the reference set. However, the algorithms are substantially different as the CPU version does not include parallel computation. CPU histograms are computed by iterating through sequences in a linear fashion and counting nucleotide types. When computing histogram for overlapping subsequences in the longer references, complete histograms are not recomputed for each window. Instead, the histogram is edited as a new character is read by incrementing the count for that character type, and decrementing the count for the character which has gone out of scope. This concept is illustrated in Figure 6.

The time recorded for the GPU implementation was substantially faster than the CPU implementation. Using the same dataset on the same machine, the GPU version completed all comparisons and results output in approximately 4m of real and user time. The CPU version required approximately 55 minutes of real and user time. This allows a speed up factor of at least 13x. Actual run-times are listed in Table 1. The time recorded for the CPU version also omits any time required for results output.

	real time	user time
GPU	4m25s	4m23s
CPU	24m39s	23m59s

Table 1: Run-times for CPU and GPU algorithms.

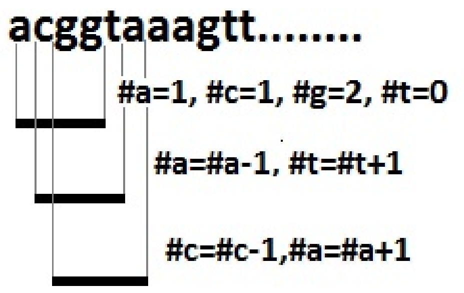

Figure 6: At the start of parsing a sequence, all nucleotide counts in the first window are stored. As the window progresses, counts are then modified by subtracting "1" from the count of the nucleotide type which has gone out of scope, and adding "1" to the type which has come into scope.

7 Summary

We describe a GPU algorithm for conducting all-against-all comparisons between two bioinformatic sequence sets. Potential string matches are located by comparing nucleotide histograms between strings and substrings in the test and reference sets respectively. The algorithms employs thrust::scan and thrust::gather functions rather than linear string parsing, thus taking advantage of the massively parallel architecture of GPU's.

We compare the run-time of the GPU algorithm to a CPU algorithm which employs linear parsing of all text sequences using a single thread. The dataset consists of 61,189 test sequences and 609 reference sequences. The GPU algorithm requires 4m25s real time while the CPU algorithm requires 55m50s. This is at least a 5.5-6x speed up factor in real and user time.

The algorithm described here is used to locate where test sequences may align with any portion of the reference sequences. Full alignment scores are not calculated, only regions which are potentially highly aligned are located. Future work will involve implementing an alignment scoring algorithm, such as the Smith-Waterman or Longest Common Subsequences to these regions after they are found to generate actual alignment scores.

References

[1] Demattè L, Prandi D (2010) Gpu computing for systems biology. Briefings in bioinformatics 11: 323–333.

[2] Elble J, Sahinidis N, Vouzis P (2010) Gpu computing with kaczmarz's and other iterative algorithms for linear systems. Parallel computing 36: 215–231.

[3] Vouzis P, Sahinidis N (2011) Gpu-blast: using graphics processors to accelerate protein sequence alignment. Bioinformatics 27: 182.

[4] Ligowski L, Rudnicki W (2009) An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases. Parallel & Distributed Processing, 2009 IPDPS 2009 IEEE International Symposium on : 1–8.

[5] Liu Y, Maskell D, Schmidt B (2009) Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. BMC Research Notes 2: 73.

[6] Manavski S, Valle G (2008) Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. BMC bioinformatics 9: S10.

[7] Walters J, Meng X, Chaudhary V, Oliver T, Yeow L, et al. (2007) Mpi-hmmer-boost: Distributed fpga acceleration. The Journal of VLSI Signal Processing 48: 223–238.

[8] Schatz MC, Trapnell C (2007) Fast Exact String Matching on the GPU. Technical report.

[9] Liu Y, Schmidt B, Maskell D (2009) Msa-cuda: Multiple sequence alignment on graphics processing units with cuda. In: Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on. Ieee, pp. 121–128.

[10] Owens J, Luebke D, Govindaraju N, Harris M, Krüger J, et al. (2007) A survey of general-purpose computation on graphics hardware. In: Computer graphics forum. Wiley Online Library, volume 26, pp. 80–113.

[11] Blelloch G (1990) Prefix sums and their applications. Synthesis of Parallel Algorithms : 35–60.