Accurate Sequence Alignment using Distributed Filtering on GPU Clusters

Reza Farivar, Shivaram Venkataraman, Yanen Li,
Ellick Chan, Abhishek Verma, and Roy H. Campbell

University of Illinois at Urbana-Champaign
{farivar2, venkata4, yanenli2, emchan, verma7, rhc} @ illinois.edu

***Abstract*—Advent of next generation gene sequencing machines has led to computationally intensive alignment problems that can take many hours on a modern computer. Considering the fast increasing rate of introduction of new short sequences that are sequenced, the large number of existing sequences and inaccuracies in the sequencing machines, short sequence alignment has become a major challenge in High Performance Computing.**

In practice gaps as well as mismatches are found in genomic sequences, resulting in an edit distance problem. In this paper we describe the design of a distributed filter, based on shifted masks, to quickly reduce the number of potential matches in the presence of gaps and mismatches. Furthermore, we present a hybrid dynamic programming method, optimized for GPGPU targets, to process the filter outputs and find the accurate number of insertions, deletions and mismatches. Finally we present results from experiments performed on an NCSA cluster of 128 GPU units using the Hadoop framework.

I. Introduction

The Next Generation of high-throughput Sequencing technologies (NGS) has fundamentally changed the landscape of the basic and applied biomedical researches. The NGS technologies are now widely applied to the transcription binding intensity measurement through CHIP-seq, RNA expression profiling through RNA-seq, DNA Structural Variation detection and Genome Assembly [1]. The modern NGS machines generate enormous amount of sequences in a span of a few hours. A fundamental problem is aligning these short sequence reads back to a reference genome. There are many algorithms that can perform this task, however a majority of them [2][3][4][5][6] are not complete, meaning that they find only one or a fixed few locations that a short sequence matches within the reference, and move on to the next short sequence. This creates a problem for the biologists trying to use the data, since the few first matching locations might not be the correct location they need to study.

In this paper we introduce a new technique to find all the possible locations within a reference sequence that match a short sequence read within a pre-specified edit distance. Our technique is based on a distributed filtering scheme which applies the pigeon hole principle to find all potential matches within the reference sequence for each read. Moreover, the distributed filter transforms the non-structured computational problem of finding all matches for each read into the reference sequence to a structured problem of pairs of potentially matching read / patterns. The structured problem can then be delegated to a hardware accelerator such as GPU to accurately weed out all false positives. In the end, our results are exact: There will be neither be a false positive nor a false negative result.

The rest of the paper is as follows. Section II introduces the problem, while table III provides the fundamental formalization of the distributed shifted masks. Section V then provides some detail about our implementation. The evaluation results are described in VI. Finally section VII provides related work and section VIII concludes the paper.

II. Gene Alignment Problem Definition

A DNA is a sequence of nucleic acid base pairs consisting of nucleotides Adenine, Thymine, Guanine and Cytosine. Gene sequencing is a chemical process that finds the order of the nucleotides in the DNA sequence and stores them in a file. The next generation sequencing techniques typically create millions of short 'reads' (or 'reads' or 'snips' in other literature), typically between 25 to 100 base pairs and in some cases (such as 454 sequencing) up to 1000. The FASTA representation of a DNA sequence (which is a simple ASCII based file format for genetic data) represents the four bases with the single letter codes A,T,G,C and the additional character N, which is a wildcard character. The N character represents an unidentified base pair in the chemical sequencing process, which means the machine could not identify which nucleotide was read at that location.

The read length we use in our implementation is 30 base pairs, however we can easily adapt the implementation to utilize larger read sizes. A next-generation sequencing machine can create up to 6 GB or around 180 million reads per day as of 2010. This number is only increasing with the newer generation of sequencing machines. The precise sequence of a long string of DNA can be queried with these shorter reads by finding the locations of these reads within the genome sequence. The reconstruction of the located reads into a larger string through the process of genome assembly will be the job of another pass (that typically require graph construction and traversal).

One main goal of the gene alignment, perhaps a few decades down the road, is to have a clear understanding of which genes (sub-sequences) in the human gene sequence are the cause of diseases. With this information at hand, and with advanced chemical gene sequencing machines of the future, a physician

can get a DNA sample from a patient and find the potential diseases that can endanger him. In such a scenario, the known set of problematic genes (reads) will be constant and the human genome sequence will be different from one patient to the next. However, currently the situation is the other way. Only a few generic human genomes have been sequenced, so that currently we have a semi-constant selection of 3 to 4 human genomes (for example typical African genome, Typical Asian genome, etc.) What is changing in current research scenarios is the read ('read') set, where researchers look for the potentially troubling genes. The choice of the fixed and dynamic data sets is important, since the fixed (or semi-fixed) data sets can be preprocessed and stored in more efficient data structures. This fact is one of the principles of the current research, as seen later in the rest of the paper. Please note that the algorithm presented in this paper is not relying on which data set is fixed, it only requires either the reads or the genome to be fixed (or semi-fixed).

The problem of matching reads into the genome sequence here is defined as follows: Given a (long) DNA sequence and a set of short reads, for each read create an output list that contains all the positions into the DNA sequence where the following 30 nucleotides after that position exactly match that read. This problem is called the "exact gene sequence alignment problem", and is a simplified sub-problem of a more complex class, "approximate gene sequence alignment". In its more complex form, the problem is to find positions into the DNA which match a read within a maximum edit distance (including insertions, deletions and mismatches), and is the problem our system targets. Note that some systems simplify the problem by finding the first (approximate) matching position within the DNA, however our aim is to find all the matches. Other systems such as Bowtie[5] solve the simpler Hamming distance problem where the only possible disagreements are mismatches at the same offsets.

We define any consecutive 30-character sequence from the genome DNA sequence as a *pattern*. A human genome sequence has 3 billion nucleotides thus contains 3 Billion minus 29 patterns. We can build the pattern set using a sliding window on the human genome. It is well known within the bioinformatics community that there is no value in considering 'plain' pattern/reads pair. These are the sequences where all 32 nucleotides are of the same type (AAAA...A, CCCC...C, etc.) Therefore these reads are filtered out and ignored. Moreover, any pattern or read that contains any N will be ignored, since N signifies an unknown value read during the chemical process, in which case there is no point in matching that read.

A mismatch is defined as unequal base pairs at the same offset in both the pattern and read. An insertion in a read (pattern) is defined as an extra base pair inserted at an offset only in the read (pattern), not the pattern (read). Likewise, a deletion in a read (pattern) is defined as a missing base pair at an offset only in the read (pattern), not the pattern (read). Note that an insertion in the pattern is equal to a deletion in the read and vice versa.

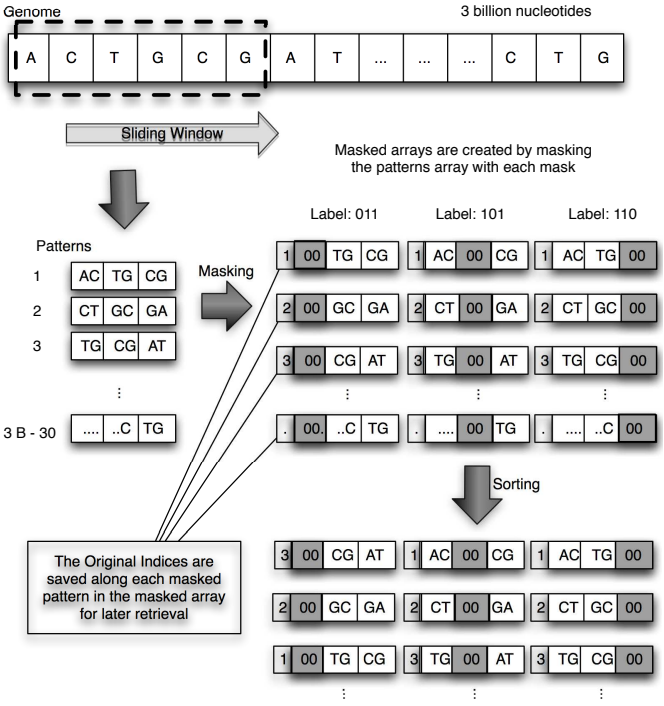

Fig. 1. Pigeon hole filter. Our proposed distributed filter shown in this figure is based on the pigeon hole principle. In this example we are looking for pattern/reads which are 1 mismatch apart. First the pattern/reads are divided into 3 divisions. The pigeon hole principle states that at least one of divisions should be exactly matching. Leveraging this fact, we can mask the divisions that might have errors and search for exact matches in the unmasked divisions. In this case there are only three ways to mask one division out of the 3: 0FF, F0F and FF0.

III. Pigeon Hole Filtering

In this section we introduce the pigeon hole filters. In section III-A we provide an incentive for the filtering. Section III-B will build the mathematical foundations of the pigeon hole filters and finally section III-C will show through a probabilistic analysis that our filter can reject the majority of random candidates.

A. The Need for a filtering phase

Consider a brute force approach comparing each pattern to all of the reads, and marking down the matching ones (accepting a maximum number of mismatches) for each read.

Theoretically, the number of operations in a brute force approach is the number of patterns multiplied by the number of reads, multiplied by the number of operations for each pattern/read pair (we will later see in section IV how a matching pair in a simplified Hamming distance problem can be distinguished in only 6 bit manipulation instructions). There are of course more instructions required to mark down the matching pairs, but considering the successful outcomes of this problem is usually very small compared to the total number of pairs, we ignore it in this simple analysis. For an example problem size of 3 billion base pair in human genome and 20 million reads, the total pattern/read pairs sum up to

$3 \cdot 2^{30} \cdot 20 \cdot 2^{20} \cdot 6 = 360 \cdot 2^{50}$ bit manipulation operators for all the pattern/read pairs. Considering a quad-core processor running at 3 Ghz, and assuming each core can work on two data sets in parallel using SSE instructions, the maximum theoretical limit on the number of bit manipulation operations is $3 \cdot 2^{30} \cdot 4 \cdot 2 = 24 \cdot 2^{30}$ per second. (Of course this is the top limit, since it is impossible in a real scenario This will translate into a runtime of $(360 \cdot 2^{50})/(22.4 \cdot 2^{30}) = 13.4 \cdot 2^{20}$ = 16,861,101 seconds = 195 days, which is too long to consider for practical purposes.

These lower limit figures of the brute force approach clearly show that a naive implementation would not be good enough for practical purposes, if feasible at all, and thus we definitely require a filtering phase, described in the next section.

B. Definition and properties

Our proposed distributed filter shown in figure 1 is based on the pigeon hole principle. Let r', r'', r''' be the total number of allowed mismatches, insertions and deletions respectively, and let $r = r'+r''+r'''$ denote them collectively as 'disagreements'. Consider a pattern and read pair each containing n nucleotides, such that their edit distance is within r. The r disagreements are distributed over the n positions and each type of disagreement has a different effect on the sequence of nucleotides. A mismatch at position i implies that the nucleotide at position i is replaced with a different one. An insertion at position i implies that an extra nucleotide is inserted in the read sequence at position i. This results in all the nucleotides from positions i to n being shifted one position to their right. Similarly a deletion at position i results in the nucleotide at position i being deleted and nucleotides originally at position $i+1$ to n are shifted to positions i to $n-1$.

Given a sequence of length n, we define a group of consecutive nucleotides as a 'division' and let the sequence be divided into $k < n$ such divisions. For simplicity, we describe equal sized divisions in the following description, but these techniques are equally applicable to variable sized divisions. Each of these divisions could contain one or more disagreements. Let r_i represent the number of disagreements in division i. We define $\delta_i = (r_0, r_1, \cdots, r_k)$ as a k-tuple representing the i'th arbitrary distribution of disagreements in k divisions, where $\sum_{x=0}^{k} r_x = r$. We can also define δ_i in more detail as $((r'_0, r''_0, r'''_0), (r'_1, r''_1, r'''_1), \cdots, (r'_k, r''_k, r'''_k))$, where (r'_j, r''_j, r'''_j) represent mismatches, insertions and deletions respectively. We will use these notations interchangeably. Δ is defined as the set of all δ_i.

Having the problem modeled this way, we can now consider the pigeon hole principle to design a filter.

Definition 1: **Pigeon hole filter.**
To determine whether a read/pattern pair are in fact within r edit distance of each other, the filter divides them into k divisions such that $k = r + p$. Then all combinations of p out of k divisions are exhaustively explored, and in each iteration the value of the selected p divisions in the corresponding pattern/read are compared to each other (ignoring the value of the other r divisions). The divisions are location aware, based on how many disagreements have happened to the left of a certain division.

Theorem 1: All read/pattern pairs whose edit distance are r will pass through a pigeon hole filter which divides them into $k > r$ divisions.

Proof: From definition 1, in a pigeon hole filter all combinations of p out of k divisions are exhaustively explored. From the pigeon hole principle we know that if we have $k = r + p$ divisions and distribute the disagreements in any arbitrary distribution among these divisions, at least p of the divisions will not have any disagreements ($r_x = 0$ for these divisions). For a read/pattern pair within r edit distance of each other there will be at least one case out of all the filters will select the p disagreement-free divisions, and for that case the p selected divisions will match. Hence, if a pattern/read pair are within r edit distance of each other, at least one of the pigeon hole filter tests will indicate a match. ■

Note that this pigeon hole filter design can have false positives, but it does not allow for any false negative. If a pattern/read pair are within r edit distance, they will pass the filter. On the other hand there can be cases when a pattern and read are more than r edit distances apart, but they can pass the filter. This case happens when more than one disagreement happens inside one division. The pigeon hole filter's guarantees are based on worst case assumptions, where the disagreements are completely spread out. Therefore an accurate second phase is needed to single out the false negative cases.

To implement the above described filter, we can construct a mask m_i directly for every δ_i. Each mask $m_i = ((q_0, s_0), (q_1, s_1), \cdots, (q_k, s_k))$ is a k-tuple such that for every r_j in δ_i, the corresponding division will be masked ($q_j = 0$) if there are disagreements in that division ($r_j > 0$), or passed-through ($q_j = F$) if there are no disagreements in the division ($r_j = 0$). Moreover, each q_j has an accompanying offset s_j representing the shifted position of that division.

To construct a mask, we define the operator $\Gamma(\delta_i) \rightarrow m_i$. The operator Γ performs as follows:

$$m_i = \Gamma(\delta_i) = \quad ((q_0, s_0), (q_1, s_1), \cdots, (q_k, s_k)) \tag{1}$$

$$q_j = \quad \begin{cases} 0, & r_j > 0 \\ F, & r_j = 0 \end{cases} \tag{2}$$

$$s_j = \quad \sum_{x=0}^{j-1} r''_x - r'''_x \tag{3}$$

where r''_x and r'''_x are the number of insertion and deletion disagreements in division x.

The number of masks found by following this procedure is equal to the size of the set Δ, which is the number of permutations that the different disagreements can be distributed across the divisions with an upper bound value of $\frac{P(n,r)}{r'!r''!r'''!}$. Obviously the memory requirement of a naive implementation will be all but impossible. However, we can reduce the number of required masks considerably, using the following lemmas.

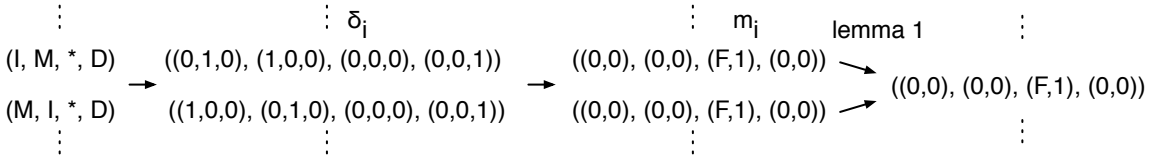

Fig. 2. Shifted masks creation. We start by considering all the possible disagreement distributions. Each disagreement distribution can be written in its k-tuple representation (δ_i). The next step is to transform each δ_i into a mask (m_i). Finally, lemma 1 can combine multiple masks.

Lemma 1: We can reduce the mask notation by setting each $s_j \leftarrow 0$ when its corresponding $q_j = 0$

Proof: This is because only the divisions whose $q_j \neq 0$ contribute towards matching and the other divisions are simply ignored. Therefore there is no information lost due to this reduction. For example, the two masks $((0,0),(0,1),(0,1),(F,1))$ and $((0,0),(0,0),(0,1),(F,1))$ can both be reduced to $((0,0),(0,0),(0,0),(F,1))$. ■

Lemma 2: $\exists \delta_i, \delta_j$ s.t. $\Gamma(\delta_i) = \Gamma(\delta_j)$

Proof: Without loss of generality, consider two sequence $\delta_i = (p_0, p_1, p_2, \cdots, p_k)$ and $\delta_j = (p_1, p_0, p_2 \cdots, p_k)$ where $p_2 = p_3 = \cdots = p_k = 0$ and $p_0, p_1 > 0$. Trivially we can see that $q_x = q_y \forall k \geq 2$ and $q_x = q_y$ for $k < 2$ by equation (2). $s_x = s_y$ for $k < 2$ by Lemma 1, and $s_x = s_y \forall k \geq 2$ since $p_0 + p_1 = p_1 + p_0$ and using equation (3). Hence $\Gamma(\delta_i) = \Gamma(\delta_j)$ ■

Definition 2: **Mask assimilation.**
We define a mask assimilation operation using the following three rules:

$$(0,0) \cdot (F,k) = (0,0)$$
$$(F,k) \cdot (F,k) = (F,k)$$
$$(F,p) \cdot (F,q) = (0,0)$$

If $m_i \cdot m_j = m_j$, the we can say that m_j assimilates m_i, denoted by $m_i \subset m_j$.

Lemma 3: If $m_i \subset m_j$, any δ_i that passes through the filter m_i will also pass through the filter by m_j.

Proof: Since m_i successfully masks the divisions of the δ_i that have disagreements, all the corresponding division locations in m_i contain $(0,0)$. From the above definition 2 the corresponding location in m_j will also contain $(0,0)$. ■

In fact the assimilation increases the false positive ratio of the filter in exchange for smaller number of masks and hence lower memory footprint.

Definition 3: **Class of masks.**
Class of p-masks, written as M_p, is defined as the set of all masks that have exactly p non zero divisions. To be able to generate this class, the number of divisions k should be at least $r + p$. To create p-masks, we start from all direct masks derived from members of Δ, and using Lemma 1, Lemma 2 and mask assimilation procedure will reduce them until all the remaining masks are p-masks.

Theorem 2: For constant values of r, the number of masks present in the set M_p is bound by a polynomial function of p.

Proof: Consider a read/pattern pair whose edit distance is within r and divided into $r + p$ divisions. The p non-zero divisions in the mask can be chosen from the $r + p$ divisions in $\binom{r+p}{p}$ different ways. For every such combination, each of the remaining r divisions have a disagreement that could either be an insertion or a deletion or a mismatch. There are a maximum of 3^r different ways in which the disagreements could be ordered. Hence the total number of possible masks is bounded by $3^r \times \binom{p+r}{p}$. As $\binom{p+r}{p} \leq (\frac{(r+p)\cdot e}{r})^r$, where e is the base of natural logarithms,$|M_p|$ is $O(p^r)$. ■

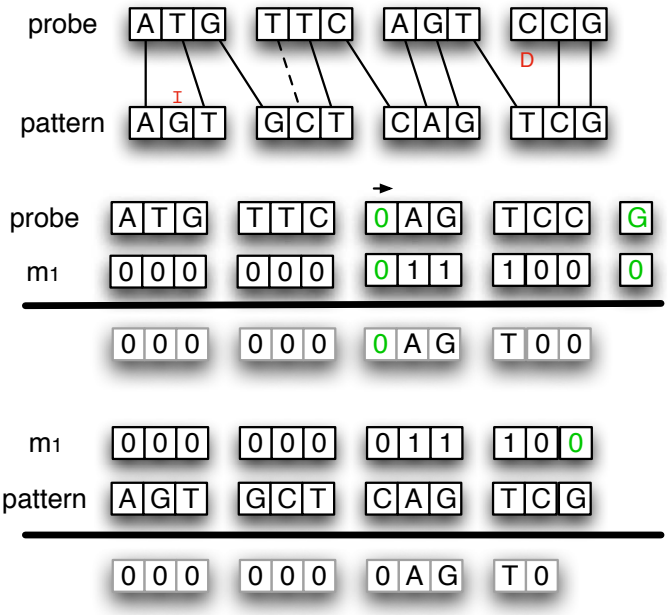

Fig. 3. Application of Shifted Masks: In this figure we show how shifted masks are applied to a read/pattern pair. There is one insertion, one mismatch, one deletion and one correct division in the sequences shown above. The pattern is masked with the shifted masks and saved into masked arrays. The read is shifted for each mask and the value is matched with the masked pattern.

C. Probabilistic Analysis on the Pigeon Hole Filter

The genomic sequences contain highly correlated data, therefore the probability of having two randomly similar sequences in a reference genome is small. However in this section we study how effectively the pigeon hole filter can

reduce the potential matches by rejecting the majority of random sequences.

The Pigeonhole Filter's rejection ratio is defined as the ratio of the read/pattern pairs that it passes over the total number of pairs it encounters as input, and is data dependent. The rejection ratio is also related to the number of correct divisions in each masked array. in this analysis, we seek to formulate the probability of the random exact matches and the expected number of random matches for the Human genome. If a reference pattern sequence is a true match to a read, it will pass the Pigeon hole Filter. However, there are also potential random patterns which might pass.

Assume that the length of the sequences in question are n and the total number of divisions are $k = p+r$. Without loss of generality and for a simpler analysis we assume all divisions are the same length $l = \frac{n}{k}$, the probability of a random pattern of length n having p non-masked divisions exactly matching their counterparts in the read is:

$$P_m = \binom{k}{p}(\frac{1}{4})^{l\cdot(p)}$$

For a given r, P_m changes drastically with different value of p. For example, if $n = 30$, $r = 2$, $p = 1$, then

$$P_m = \binom{3}{2}(\frac{1}{4})^{\lceil\frac{30}{3}\rceil\cdot(1)} = 3\cdot(\frac{1}{4})^{10}$$

Therefore for a given read and assuming that the all $N_p \approx 3\times 10^9$patterns in the genome reference sequence are completely random, the expected number of matched patterns would be:

$$Exp(\text{random matches}) = N_p \cdot P_m \approx 8583$$

We can see that there are a lot of random exact matches when $p = 1$. However, if p is set to 2,

$$P_m = \binom{4}{2}(\frac{1}{4})^{\lceil\frac{30}{4}\rceil\cdot(2)} = 6\cdot(\frac{1}{4})^{16}$$

and the expected number of matched sequences is reduced to

$$Exp(\text{random matches}) = L_h \cdot P_m \approx 4.19$$

Therefore we see that by using proper settings for pigeon hole filter, the majority of the random matched sequences are filtered out, saving a large number of unnecessary computation in the Post-Filter processing. In a following section we will show how the filter's rejection ratio responds to the changes of p when using real data.

IV. Post-Filter Sequence Matching

Once the input patterns are filtered for the reads and a set of potential matching patterns are found, we need to do a more thorough analysis and find out the exact number of insertions, deletions and mismatches. In other words, the filter might have passed a number of masked pattern/read pairs whose number of disagreements are more than the required threshold, which can happen if more than one disagreements occurred in one divisions. In the post-filter phase, the exact number of gaps and mismatches is identified and the run-away cases are rejected. While weeding out the impossible matches, the filter has an interesting side effect as well. It in fact puts structure on an unstructured data set. Having a structured set of data past the filter enables us to look into data-parallel SIMD accelerators as possible candidates to perform the heavy processing work. In our implementation we decided to leverage the massive parallelism potential of GPU accelerators, which we will talk about in detail in a later section.

The best known way to accurately determine the edit distance of two sequences is through dynamic programming. To find the best alignment of the two sequences, we need to perform global alignment. A general global alignment technique is the Needleman-Wunsch algorithm [7], which was the first application of dynamic programming to biological sequence comparison. There have been other variations of the dynamic programming solutions in literature [8], [9]. In [7] to find the alignment of two strings A and B, a two-dimensional matrix is allocated. As the algorithm progresses, the (i, j)'th entry of the matrix will be assigned to be the optimal score for the alignment of the first i characters in A and the first j characters in B. This value will be computed based on the i'th character in A, the j'th character in B and the alignment scores at $(i-1, j-1)$, $(i, j-1)$ and $(i-1, j)$ which are already computed and stored in the matrix. The algorithm also keeps track of how the maximum alignment was selected for each cell (up, left or diagonal) and stores them in a second shadow matrix. Once the matrices are computed, the bottom right hand corner of the matrix will contain the maximum score for any alignments. To compute which alignment actually gives this score, we start from the bottom right cell, follow the directions stored in the second matrix to construct the alignment.

The runtime and memory requirement of [7] are both $O(m \cdot n)$. This rather high memory usage makes it unsuitable for the SIMD cores of a GPU, since the amount of fast shared memory per computing core is quite limited. There are other variations of [7] based on the longest common subsequence (LCS) implementation presented in [8], where they try to reduce the memory usage. In [8], the regular dynamic programming LCS problem is mixed with a divide and conquer strategy, based on the principle of optimality, to reduce the memory footprint of the algorithm to $O(min(m, n))$ while the run-time stays in the same order of $O(m \cdot n)$. In practice this algorithm requires about an order of magnitude more computation, however the algorithmic order remains the same.

A variation of [8] has been proposed in a slightly different way in [9], and our work is a variation of this algorithm. In [9] the matrix is divided into 4 quadrants. The pivot point for division can be conveniently placed on the center of the matrix, although there is no theoretical obligation to do so. The algorithm takes a divide and conquer approach and divides each quadrant until the point that the alignment in each quadrant becomes 'trivial'.

To find the alignment inside a quadrant the algorithm only needs the left and top boundaries of the quadrant, as well the corresponding subsets of the original strings. Note that

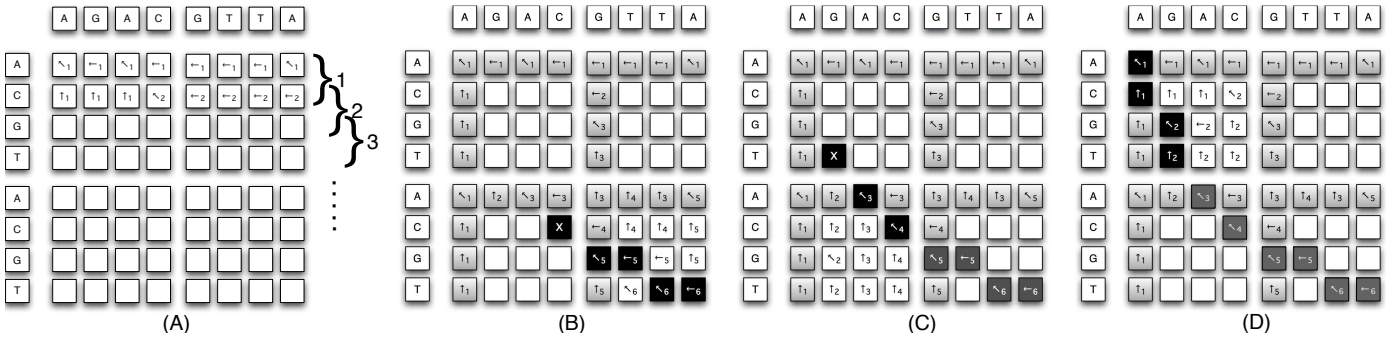

Fig. 4. Accurate alignment computation in the GPU. A) The first pass of the algorithm keeps only two active rows of the alignment matrix while scanning it from top to bottom. During this scanning pass, it computes the boundary values of the smaller trivial quadrants for later access by the second pass of the algorithm, shown as shadowed cells in (B). B) The second pass of the algorithm relies on the boundary values calculated in the previous pass. Having these values ready for each quadrant, we can start from the last quadrant and compute the inner values using a simple Needleman-Wunch dynamic programming variant. We then start tracking back from the last element of the matrix and follow the directions to find the exit cell, denoted by letter 'X'. (C) Keeping a record of the trace-back so far, it is continued in a new quadrant using the exit value of the previous quadrant. (D) The algorithm finally exits the larger alignment matrix through a quadrant either on the left edge or top edge of the alignment matrix.

as long as the boundaries are the only goal, they can be computed using a simplified form of [7] that only keeps two consecutive rows of the matrix for storage space, using $\Theta(2 \cdot m)$ memory. Therefore, the algorithm calls the boundary calculation function for each trivial quadrant and finds the optimal alignment in that quadrant. Then it does the back-tracing, following the 'directions' and finds where the global alignment path exits this specific quadrant. It then passes the exit point as entrance point to the next quadrant.

Our algorithm is an adaptation of [9], in which the recursions are replaced with loops. Our implementation is however hand-tuned for best performance on current generation GPUs. Starting with an analysis on the available resources of the GPU to maximize its utilization, we find the size of a 'trivial' quadrant, which in our current implementation is set to 4 by 4. As such, we divide the large matrix required for global alignment into 64 quadrants, numbered from $(0, 0)$ to $(7, 7)$ based on their location in the alignment matrix. The trivial quadrant problem is solved with a dynamic programming variation of [7] in $\Theta(\frac{m}{4} \cdot \frac{n}{4})$ run time and $\Theta(\frac{m}{4} \cdot \frac{n}{4})$ memory.

Similar to [9], the first step to process a trivial quadrant is to compute its top and left boundaries. As depicted in part (A) of figure 4, each time the boundaries of a trivial quadrant is required a pass on the alignment matrix is performed and the required boundaries are passed along for the trivial quadrant calculation. The memory requirement for this pass is twice the size of each sequence.

The next pass of the algorithm starts from quadrant $(7, 7)$. In this pass, the entrance point of the trace-back route is set to the last element of this quadrant, which is also the last element of the larger alignment matrix. From here, after calculation of the alignment in this trivial quadrant using the pre-computed boundary values, the exit point on the trace-back path is found in another quadrant. This new quadrant is either on the left, top or top left of the previous quadrant, as the alignment trace-back route is monotonically increasing from bottom right of the alignment matrix to its top left. This pass of the algorithm is depicted in parts $(B), (C)$ and (D) of figure 4 (albeit shown for a simpler case).

A. Final Phase

In the last phase, the program creates a list of pattern locations (chromosome_id:index_into_chromosome) into the genome that approximately match (within the allowed limits) each original read, along with the number of mismatches found in that particular approximate match. The output is generated using the SAM format, including correct CIGAR strings for easy verification. Because the masks are distributed across a cluster, it is possible that a truly matching patter/read pair are identified in each machine they run in. Therefore duplicate pattern/read matches will exist after the second phase across the cluster. To consolidate the results a simple reducer is run across the cluster that unifies the results and writes them to disks.

V. Implementation

We implemented the proposed algorithm on a cluster of GPU-enabled computers. The open source Hadoop project, an implementation of the MapReduce programming paradigm, was selected. To be able to run Hadoop on the target cluster which is mainly configured for batch jobs, we utilized a special Hadoop virtual provisioning system called 'Hadoop on Demand' (HOD), that uses the Torque resource manager to do node allocation. On the allocated nodes, it can start Hadoop Map/Reduce and HDFS daemons. Once in Hadoop, we use 'Hadoop streaming', which lets the programmer to write two program, one for "map" and another to act as "reduce", in a language of his choice and provide the binary files to the framework to be executed in a distributed manner. The input stream is still provided by the framework, and each program's output to the standard-out will be parsed by the framework as either intermediate key/value pairs in case of the mapper or the final results ready to be written on the disk for the case of the reducer. The choice of Hadoop streaming was

mandatory to allow integration with the CUDA framework for GPGPU programming. The whole program is implemented and evaluated on the NCSA's AC cluster, consisting of 32 nodes. Each node has Two dual-core 2.4 GHz AMD Opterons, 8 GB of memory and one NVIDIA Tesla S1070 containing 4 GT200 GPUs, each with 4 GB of memory.

A. Distributed Filtering

The central key to implementing the distributed filtering scheme is to find the right set of masks and distribute them across the computing nodes of the cluster. Once the masks are found, each 'mapper' program creates its corresponding set of masked arrays in the memory and starts processing through the reads one by one. If any read after being masked (and shifted in the process) can be matched in a masked array, it will be inserted in a buffer along with the matching pattern for further processing. Because GPUs are take a considerable time to start processing a computing job, it is not efficient to send a matching pair to it as soon as they are found, hence the buffering. In our current implementation the buffer can contain 2 million pairs before being dispatched to the GPU.

B. Post-Filter Sequence Matching

The implementation of the algorithm described in section IV involved many optimizations required to reduce the memory usage of each thread. Since the amount of computation per each data input (and eventually output) is quite considerable, the computation is not memory bound, therefore we thrive to increase the utilization of the GPU to maximize the performance of this algorithm. We can calculate the maximum amount of register and shared memory available to the program for each thread for a certain device occupancy. Balancing the resources and the occupancy proved to be a challenging optimization task. Eventually we managed to balance the occupancy and requirements in a way that the device works at 62.5% occupancy, while each thread uses 25 registers and 15 bytes of shared memory. With more optimization we can theoretically increase the device occupancy to 100% by reducing the register usage to 16 registers. However we decided to stop optimizing the program at 62.5% occupancy since it was already in a balance with our Distributed masked filtering implementation.

VI. Evaluation

We evaluated the system on the NCSA AC cluster, using a set of 7 million reads containing 30 nucleotides each, and processed them against the NC_123456 Genomic Mixed human genome sequence reference, which contains around 3 billion nucleotides within 21 chromosomes. This section describes the evaluation results.

A. Distributed Filtering

As mentioned earlier, the total number of masks has a polynomial relationship to the number of correct divisions in the filter. Increasing the number of correct divisions for a fixed number of mismatches increases the accuracy of filtering. In the limit, if we set $(p + r) = n$, we can surely know that this pattern should be an acceptable pattern, and the exact location of the mismatches will also be known if any division has at most one disagreement. However, increasing the number of divisions increases the computational power and memory requirements for the fine grained matching. Therefore we need to find an acceptable tradeoff based on how much processing power and memory is available, and let the second phase of computation pick the correct set of matches out of the potential acceptable results passed by the filter (which will surely have false positives).

Table I shows the total number of masks for a select number of configurations. For a constant edit distance, the total number of masks is small enough to run the whole distributed filter on our 32-node cluster.

TABLE I
NUMBER OF UNIQUE MASKS REQUIRED

Mismatches	Ins	Dels	Correct Divisions	Unique masks
1	1	1	1	10
1	1	1	2	40
1	1	1	3	104
1	1	1	4	215
1	1	1	5	386
2	2	2	1	31
2	2	2	2	346

Table II depicts the efficiency of the filter in a few configurations which we experimented with.

TABLE II
PERCENTAGE OF PROBES THAT PASS EACH FILTER

Filter Passed Percent	Avg. number of patterns matched	Mask used
85.38	13.6	1-1-1-2
13.39	21.38	1-1-1-3
2.84	46.66	1-1-1-4

To measure the effectiveness of our GPU implementation against a traditional CPU, figure 5 shows the runtime on a CPU, GPU 1 dimensional block and GPU 2 dimensional grid.

VII. Related Work

Due to the rapid development of next generation sequencing technology, billions of sequence data is generated every day. One of the fundamental problems is to map the massive amount of short sequences to the genome. Sequence alignment is the process of scanning a reference genome for matches to a small subsequence or read. This poses a great challenge to the Bioinformatics community to develop algorithms that can align large number of short sequences to the genomes in an efficient way. To address this problem, a number of several algorithms have been developed. They can be classified as a variation of two fundamental technologies for short-read sequence alignment: (1) Seed and Extend; (2) Burrows Wheeler transform (BWT)-based methods.

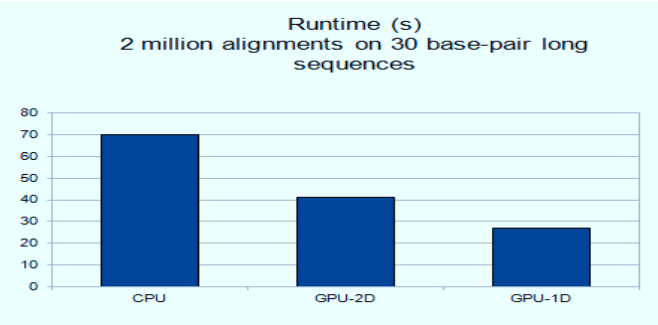

Fig. 5. Runtime on CPU vs GPU

Several tools identify the exact matching seeds and then extend to the final results. ELAND[2], RMAP[10], ZOOM [**?**] and SeqMap[3] index the reads, while SOAP[4], BFAST and MOSAIK index the whole genome at the expense of more memory usage. ZOOM[11] builds a hash table of the reads using the spaced seeds in which 1s represent matching positions and 0s representing irrelevant positions. A new generation of BWT-based approaches, including BOWTIE, BWA [6] and SOAP2, have gained much attention recently. They are built on the compressed suffix array data structure call FM index, which has two advantages: First, it can do subsequence search very efficiently – the runtime performance of these methods usually outperforms other methods by 10 folds. Secondly, the final index for the human genome only requires about 2.3 GB in size, which make it possible to store the whole index of a big genome in main memory.

Although these algorithms are effective to the short sequence mapping. They are all designed to be run in a single process, or a naive multithreading model in a multi-processor machine. As the scale of data increases, parallel algorithms designed to run on multiple machines become promising. To the best of our knowledge, CloudBurst [12] and Crossbow[**?**] are the only two program using the MapReduce framework to parallelize the short sequence alignment. However, CloudBurst requires a huge amout of disk space in the Reduce phase. Also, it takes a long time to generate the outputs, since a large number of intermediate results are generated. Although Crossbow [**?**] runs faster, it aims at finding Single Neucleotide Polymorphism by employing a naive implementation with the Hadoop framework on top of the Bowtie algorithm.

Our research is also related to the sensitive pair-wise alignment problem which has been previously addressed using global and local alignment techniques. For example, the Needleman-Wunsch algorithm [7] is a general global alignment technique which is based on dynamic programming, with computational complexity of $O(m \cdot n)$, where m and n is the length of the two sequences respectively. On the other hand, the Smith-Waterman[13] algorithm is a general local alignment method addressing the problem of finding local similarity segments in two dissimilar sequences. This algorithm is also based on dynamic programming with time complexity of $O(m \cdot n)$.

Recently, GPU computation techniques are employed to speed up the local and global pair-wise alignment methods. For example, MummerGPU [14], [15] uses suffix tree based algorithm on GPUs and provide 13x performance speedup. Striemer et al. [16] implement the Smith-Waterman algorithm for local alignment on GPUs and achieve a 23x speedup. Liu et al. [17] tackled the problem of protein similarity using biosequence database scanning on GPUs implementing the Smith-Waterman algorithm.

VIII. Concluding Remarks and Future Work

With the growing importance of Next Generation Sequencing technologies, fast sequence query systems are necessary to handle the growing volume of information collected. Unlike existing bioinformatics tools which use imprecise alignment algorithms to achieve satisfactory speed, we introduce a novel 2 phase algorithm that performs accurate sequence alignment by distributing filtering using the pigeonhole principle and a cluster of GPUs.

References

[1] A. Kahvejian, J. Quackenbush, and J. Thompson, "What would you do if you could sequence everything?" *Nature biotechnology*, vol. 26, no. 10, pp. 1125–1133, 2008.
[2] A. Cox, "Eland - Efficient Alignment of Local Nucleotide Data," Unpublished. [Online]. Available: Unpublished.
[3] H. Jiang and W. Wong, "SeqMap: mapping massive amount of oligonucleotides to the genome," *Bioinformatics*, vol. 24, no. 20, p. 2395, 2008.
[4] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, p. 713, 2008.
[5] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.
[6] R. D. Heng Li, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics (Oxford, England)*, vol. 25, no. 14, pp. 1754–1760, July 2009.
[7] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, March 1970.
[8] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, 1975.
[9] R. A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," in *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. New York, NY, USA: ACM, 2006, pp. 591–600.
[10] A. Smith, Z. Xuan, and M. Zhang, "Using quality scores and longer reads improves accuracy of Solexa read mapping," *BMC bioinformatics*, vol. 9, no. 1, p. 128, 2008.
[11] H. Lin, Z. Zhang, M. Zhang, B. Ma, and M. Li, "ZOOM! Zillions of oligos mapped," *Bioinformatics*, vol. 24, no. 21, p. 2431, 2008.
[12] M. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, p. 1363, 2009.
[13] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[14] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg, "Versatile and open software for comparing large genomes," *Genome biology*, vol. 5, no. 2, p. R12, 2004.
[15] C. Trapnell and M. Schatz, "Optimizing data intensive GPGPU computations for DNA sequence alignment," *Parallel Computing*, 2009.
[16] G. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," *IPDPS, May*, 2009.
[17] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 8.