FastHASH: A new GPU-friendly algorithms for fast and comprehensive next generation sequence mapping

Hongyi Xin Donghyuk Lee
hxin@cs.cmu.edu donghyu1@andrew.ece.cmu.edu
Carnegie Mellon University

Abstract

The advent of Next-generation sequencing (NGS) makes it possible to produce DNA sequence information effectively by dividing individual DNA to massive number of short fragments and reading these massive sequences concurrently. At the result of these parallelized processing, high computational resources are needed to reconstruct the divided fragment sequence as the original order. mrFAST, one of the DNA sequence matching tools, promise highest comprehensiveness, however, it is slower than other tools. We propose a new algorithm, FastHASH, which drastically improves performance over mrFAST, while maintaining comprehensiveness.

1. Introduction

1.1. Next-generation DNA sequencing

For a long time, many researchers have tried to find out the relationship between DNA and other characteristics, like species, ethnic group and individual specific characteristics, i.e. hereditary diseases. As the result of these studies, a lot of useful characteristic information determined by individual DNA are discovered, which can be applied to individual medical services. However, the high cost of individual DNA sequencing would makes it difficult not only to use DNA information for individual medical broadly, but also to accelerate the development of DNA related studies. Next-generation DNA sequencing techniques [12, 13, 4, 14, 5] are introduced to make it possible to reduce the DNA sequencing cost drastically by analyzing individual DNA as massively parallelized manners, like Roche/454 FlX Pyrosequencer [**?**], Illumina Genome Analyzer [2] and Applied Biosystems SOLiD Sequencer [1]. The main features of these techniques are 1) to divide individual human genome as small fragments of 100 - 200 base pairs length and 2) to analyze them concurrently. So, by these parallelized methods, the DNA analyzing cost can be drastically reduced. However, after that, the massive DNA sequence data divided to short sized fragment have to be reconstructed as the orders of original sequence. The possible way to reconstruct the divided individual DNA fragments is to match them to Reference DNA sequences, already constructed and verified the correctness by previous studies. This reconstruction method is available because the DNA differences or variations between individual humans or living creatures within same species are small. So, Next-generation sequencing technologies provide low-cost and high-throughput genome sequencing at the cost of requiring enormous computational resources, i.e. matching 31.5millon fragments that each fragment has 100 base pairs, to 3.15 billion sized Reference sequence with allowing several mismatches or indel (insertion or deletion of gene). Now, one of the key challenging points of Next-generation DNA sequencing techniques is how we can match these massive fragments to Reference sequence cost-efficiently.

1.2. DNA Sequencing tools

For last several years, many tools for DNA sequence matching are introduced [9] and they can be divided to two groups by their characteristics. The one group of tools is more focused on the matching speed, based on the Burrows-Wheeler-Transform (BWT) (Burrows and Wheeler, 1994) [3] and Ferragina-Manzini index [10] such as BWA (Li and Durbin, 2009) [11] and SOAP2 (Li et al., 2009) [7]. These tools are fast for searching for exact matching. However, they are not comprehensive, i.e. they do not search for all possible locations of each fragments. The other approach puts emphasis on the comprehensiveness based on hash table based seed-and-extend algorithms such as mrFAST (AlKan et al., 2009) [6] and SHRiMP(Rumble et al., 2009) [8]. These tools are comprehensive, however they are typically slower than the tools of the formal approach. Both of these two matching algorithms branches are trying to make their algorithms to be compatible to Single-instruction Multi-data (SIMD) architecture or General-purpose graphic processing units (GPGPU) to increase matching speed.

1.3. New Algorithm: FastHash

In this work, we proposed a new algorithm, FastHASH, which drastically improves performance over mrFAST [6], while maintaining comprehensiveness. Our key observation is that mrFAST performs too many costly computations that can be avoided by a smart algorithm. For every read, mrFAST will first find out all potential locations in reference genome. Then for each possible location mrFAST performs an expensive string comparison function to extract complete differential information between the input read and the reference genome, even if the location will not match. We propose two key ideas to reduce both type of computation. First, we drastically reduce the number of potential locations considered for string compare function, while still preserving comprehensiveness. We call this method Cheap Key Selection. Second, we drastically reduce the number of string comparison performed by rejecting obvious incorrect locations in the early stages of mapping. We call this method Adjacent Filtering. Our initial CPU implementation of FastHASH provides 14-fold speed up over mrFAST, while still preserving comprehensiveness. Since FastHASH does not diverge, which is essential for GPU implementation, it is also compatible on GPUs.

In the next section, we describe the matching algorithms and the characteristics of representative matching tools. In section 3, we present the mechanism of FastHash in detail. In Section 4, we describe the mechanism of our gpu implementation. We also describe our test environment and methodology in secton 5. In Section 5. Finally, we conclude with the performance of FastHASH compared with mrFAST in Section 6.

2. Background

Several mechanisms have been proposed for next generation sequence alignment. They can be summarized into two categories: Burrows-Wheeler transform based aligning tool and hash-table based aligning tool. Burrows-Wheeler transform based aligning tools usually can search for exact match very fast while slow with inexact matches (matches with the presents of errors). Hash-table based aligning tools are slow in general but they promise to find all possible locations in reference DNA sequence where the reference DNA string has fewer than e number of errors. We will briefly describe how they work by showing one example implementation for each category.

2.1. Burrows-Wheeler transform based aligning tools

Burrows-Wheeler transform, together with backward search, is able to mimic the top-down traversal on the prefix tree of the reference gnome with relatively small memory footprint. The time complexity of finding the exact match is O(m), with m being the fragment length. For inexact match, such implementations normally will have to traverse neighbor sibling nodes at every level of the tree and their corresponding descendants. Such branching traversing behavior causes the complexity of inexact match to be roughly O(me). To further explain how it works, let us look at one typical tool BWA. To understand BWA, we will have to first understand how prefix tree and backward search works. Then we will explain how Burrows-Wheeler transform and backward searching can be used to solve DNA sequencing problem.

2.1.1. Prefix tree and string matching

The prefix tree of string X is a tree where each edge is labeled with a letter and the string concatenation of the edge letters on one path from a leaf to the root gives a unique prefix of string X. With the prefix tree, testing whether a query W is an exact substring of X is equivalent to finding the path starting from root and reconstructs W along the way. Such testing procedure can be done in O(|W|) (|W| is the length of W) time by matching each letter of W to an edge, starting from root. For inexact match, instead of matching letters for all edges along the path, now there will be at most e different edges allowed for one traverse from leaf to root. In order to achieve such error endurance and collect all potential paths, mismatches as well as deletion and insertion will have to be allowed at each step traversing the tree, which means not only the exact match edge will be considered traversing down, but also sibling not matching edges will be traversed for completeness, we call this full traverse. However, for each different edge ever passed, the error counter of such path will be incremented, and the traverse will stop whenever the error counter exceeds the error threshold e.

2.1.2. Burrows-Wheeler transform and backward searching

Burrows-Wheeler transform is a string transformation function, where for an input string X it first rotates the text and stores every rotated string in a matrix. Then, it sorts all rotated string in lexicographic order. The transformed result string B of string X will be the right most column of the matrix. There is also a sorted rotation index vector S[], where S[i] denotes the original order of the now sorted ith rotated string. There is an important property associated with Burrows-Wheeler transformed string. To explain the that, we first define:

$$R_{min} = min\{k : \, W \, is \, the \, prefix \, of \, X_{s[k]}\}$$
$$R_{max} = max\{k : \, W \, is \, the \, prefix \, of \, X_{s[k]}\}$$

Above equations translate to *R_{min}(W) is the smallest order k of the rotated string of $X_{s[k]}$ where W is the prefix of $X_{s[k]}$. Rmax(W) vice versa.* If there is just 1 single exact match, R_{min}(W) = Rmax(W). Let C(a) be the number of symbols in X that are lexicographically smaller than a and O(a, i) be the number of occurrences of a in Burrows-Wheeler transformed string B[0, i], Ferragina and Manzini proved that:

$$R_{min}(aW) = C(a) + O(a, R_{min}(W) - 1) + 1$$
$$R_{max}(aW) = C(a) + O(a, R_{max}(W) - 1)$$
$$R_{min}(aW) <= R_{max}(aW)$$

Above equations translated to *given R_{max}(W) and R_{min}(W) of string W, we can get the R_{max} and R_{min} for a new string aW where aW is the original string W with 1 more letter "a" attached in front of it.* Ferragina and Manzini further proved that these 3 condition holds if and only if aW is also a substring of X. Given these 3 equations, together with pre-calculated values of C(a) and Burrows-Wheeler transformed string B, one can quickly get the one of exact match location by picking either R_{min} or R_{max} and get their corresponding S[R_{min}] and S[R_{max}]. The way it calculates R_{min}(aW) and R_{max}(aW) given R_{min}(W) and R_{max}(W) is exactly a mimic of traversing the prefix tree. For exact match, BWA saves memory since they do not need to store the prefix tree but just a Burrows-Wheeler transformed string B and some pre-calculated values of C(a) where a will be any of the for base pairs {A, C, T, G}. However, when matching for inexact case, BWA will also need to mimic full traversing of the prefix tree by also calculating not only R_{min}(aW) and R_{max}(aW) for each additional letter a, but also R_{max}(bW) and R_{max}(bW) where b can be all the other mismatched base-pair. The branching behavior will become even worse when taking insertions and deletions into account. As a result, BWA performs poorly when trying to get all possible locations.

2.2. Hash-table based aligning tools

Hash-table based DNA aligning tools on the other hand, store all locations of reference DNA and the corresponding pattern at the location in a hash-table, where the pattern is the key and the corresponding locations are contends associated with each key (the locations is also called coordinates they are inter-changeable in this paper). For each aligning process, generally a hash-table based aligning tool will first break the fragment into several small segments and then use the segments as keys to look into the hash-table. The hash-table returns all locations that the query pattern associates so that the tool can retrieve fragment-size long reference DNA sequence at each location. Finally there will be a string comparison test, which determines the similarity between input fragment string and the reference strings at each location returned by hash-table. Only the locations that pass the string comparison test with a high confidence score will be considered a potential match up. Among all hash-table aligning tools, we analyzed mrFAST, which is a widely used tool guaranteeing full coverage of all possible locations within some error number *e*.

2.2.1. Hash-table query

The first step of aligning process for mrFAST is cutting an input fragment into several small segments and performing the hash-table look up. For exact matches, picking any segment as key to access hash-table would guarantee full coverage of all exact match locations, since the coordinates stored inside the hash-table matching one segment is the super set of the coordinates matching the whole fragment (the whole fragment can be matched only if the segment is matched). For inexact match, if we allow e errors, then picking e + 1 segments as keys will cover all potential locations, since at least 1 key will guarantee to be error free, by Pigeon Hole theory (If there is m+1 pigeon and m holes, at least one hole will have 2 pigeons). Normally, researcher will allow around 5% of the fragment as errors. With a pattern length (key length of the hash-table) of 12 base pairs, the worst case would be using half of the segments as keys. If however, the error number is larger than the segments number, the pigeonhole theory will no longer hold, simply because there are not enough segments. To solve such problem, one can shrink the segment (key) length thus force producing shorter but more segments. Notice that this will decrease the length of the keys and causing more coalescing and longer coordinate list in the hash table. As we will see later, this may degrade the performance as now each key will have a longer coordinate list to traverse, which results in more string compares.

2.3. String Compares

There are lots of string compare functions. SARUMAN uses Needleman-Wunsch algorithm, while mrFAST uses Ukkonens edit-distance calculation algorithms. The detail of edit-distance calculation algorithm is out of the scope of this paper. But in short, it returns the exact number of mismatches, insertions and deletions as well as their locations and different contents. While providing such detailed information between input fragment string and reference DNA string, edit-distance calculation is extremely computationally expensive. As a result, we should avoid calling edit-distance calculation if the coordinate can be rejected at early stage, before edit-distance calculation. In short, mrFAST provides full coverage for all possible matching locations given an max error number e. However, due to large amount of edit-distance calculations and their intrinsic expensive property, mrFAST suffers from long execution time.

2.4. FastHASH

As mentioned earlier, Burrows-Wheeler transform based aligning tools, or any other suffix tree or prefix tree based aligning tools are fast for exact match but will get exponentially slower as the number of error tolerance increases. On the other hand, the computation complexity of hash-table based DNA aligning tools grows linearly as error tolerance increases but they are slow in general. Our goal is to provide a fast but comprehensive DNA aligning tool. By comprehensive we mean promise full coverage for finding all potential locations where the number of differences between input fragment and reference DNA string is lower than a user set error number e. To preserve this comprehensive property while avoiding exponential slow down as error rate increases, we decide to optimize hash-table based aligning tool mrFAST.

3. Algorithm Optimization

3.1. Adjacency Filtering

mrFAST suffers from long execution time for several reasons. One of the biggest reasons is that the edit-distance calculation is time consuming. The other reason is that the hash-table is not balanced. By not balanced we mean some patterns store more coordinates within its entry whereas some patterns store fewer coordinates. For those patterns storing long lists of coordinates, we call them expensive keys. Expensive keys imply two problems: 1. Expensive keys have long coordinate list, which means if ever used them as keys, they will impose many string comparisons and hence many reference DNA database accesses. 2. In real test cases, they will be frequently encountered. This property is associated to the first propertythey are popular in human DNA so that they have longer coordinate list. As a result, when sampling, we will have higher chance encounter these expensive keys since they show up more frequently in human DNA.

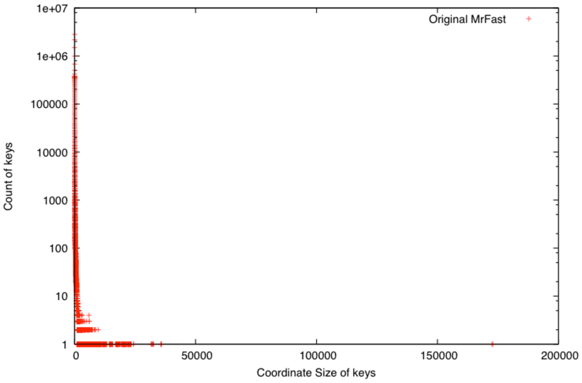

Figure 1. The coordinate entry size

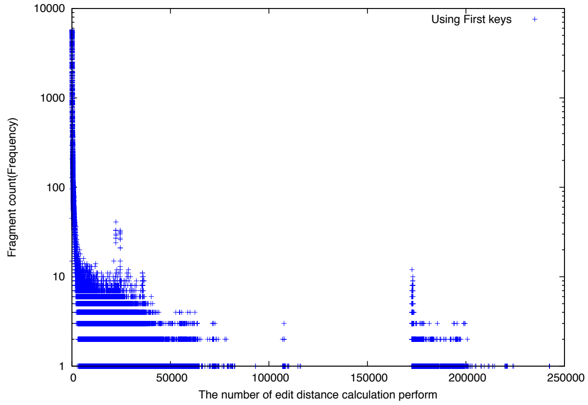

Figure 2. The number of edit-distance performs

Figure 1 shows the coordinates entry size for chromosome 1. For a vast majority of the entries, there is 0 coordinate stored inside which means this pattern never shows up in this chromosome. However, there are also some coordinates having more than thousands or even millions of coordinates. As a result, whenever mrFAST uses those segments as keys, there will be thousands to millions of edit-distance conducted. Figure 2 shows how many edit-distance calculations performed turned out to be a match. When aligning 10 million fragments from chromosome 1 to chromosome 1, allowing max error number to be 3 normally out of 3000 edit-distance calculations, 1 will be a match. Such low matching rate implies that the majority of the coordinates subject to edit-distance calculation will be rejected, at a high execution time cost. From our analysis, it turns out some of the not matching coordinates can be rejected solely by analyzing the information from the hash table alone, without even accessing reference sequence database and performing the expensive edit-distance calculation. Such early stage rejection would save memory bandwidth, on chip cache as well as execution time.

The key observation behind this early rejection is that: if the fragment can be divided into N segments, and we probe e+1 segments coordinate lists as we described in 2.2.1. Then for each coordinate stored in the coordinate list, if such coordinate is a potential match, the adjacent keys have adjacent locations stored in their coordinate lists. An example will be the best explanation. For instance, as shown in Figure [3a?], a fragment f is divided into N segments s1, s2 sN, with each segments being L in length. When probing the coordinate entry for the first segment, on the first coordinate coor1 from s1, we test if coor1 + L is located in the coordinate list of s2 and if coor1 + 2*L is located in the coordinate list of s3, so on and so forth. For a perfect match, all adjacent segments should locate at corresponding locations, which means all corresponding locations should be stored inside the corresponding coordinate list. However, for inexact matches, where we allow mismatches, the constrain will be relaxed to at least N e segments should find corresponding locations in their coordinate list. The reasons why we relax the filtering constrain from all segments to N e segments is that now there could be at most errors in the fragment, which in the worst case could be distributed in e segments. In presents of insertion and deletions, the filtering test will be further relaxed to the range of expected coordinate plus or minus e. For example in Figure [3b?], if now the error number e is set to 3, then for coor1, instead of searching for coor1 + L in the coordinate list of s2, we now search for coor1 + L +- e for s2 and coor1 + L*2 +-e for s3, so on and so forth. Additionally, instead of requiring matches for all of the segments, the fragment will pass the filtering test if more than N e segments found corresponding locations.

Adjacency Filtering itself will not guarantee matches but it will filter out obvious not matching locations. If a fragment has more than N e segments that fail finding adjacent coordinates, we can deduce there must be more than e errors thus rejecting the coordinate. However, if the fragment passes the adjacency filtering, this does not necessarily mean the input fragment matches to the reference DNA sequence within e errors. We cannot guarantee the total error number is less than e since for those segments that failed finding adjacent locations in their coordinate lists, they might have more than 1 errors in it. Thus the adjacency filtering itself is not complete. For a matching input fragment and reference DNA coordinate pair that passes adjacency filtering process, we still have to perform edit distance calculation on them to get the exact number and locations of the errors. Although adjacency filtering is not complete and cannot replace edit-distance calculation, it did drastically reduced the times of edit-distance being called. We will show the numbers in evaluation section.

3.2. Cheap Key Selection

Pigeon Hole theorem and multiple search keys There are several reasons why we have to allow several errors for finding out potential matching locations. First is the potential difference between individual DNA sequences. Most portion of individual DNA sequence is similar with Reference DNA sequences and using this similarity, we have tried to reconstruct individual DNA sequences by matching it to the known Reference sequences. However, most of important portions of DNA sequences are the difference between DNA sequences. These differences can cause specific diseases or determine specific individual characteristics. So, one of the most important thing for matching fragment to reference sequence is to give the potential location within maximum allowable errors, such as insertion, deletion and mismatch. Second reason is that there are potential errors caused by DNA sequencing analyzers misreading. For example, Illumina platform has relatively poor performance in assorting G-C, i.e. Illumina platform frequently misread G as C or vice versa. The other NGS platforms also have their own sequencing biases. By matching fragments to Reference sequence with allowing specific number of errors, we can reduce the matching failures caused by NGS platforms specific errors. String comparison operation of mrFAST can give the edit-distance between two sequences, which can be the threshold values to decide whether the location can be the potential location of fragment or not. However, main assumption of hash table based seed-and-extend algorithm is that there is no error in the fragment of key. If we assume that the fragment has some errors caused by a certain reason, like individual DNA difference or platform biases, and the errors are located at the searching key for hash table, then, the coordinates according to the search key have potential errors. Finally, we lose the chance that we can find out exact matching cases with allowing these errors.
The obvious solution of this potential problem is to select multiple search keys within the fragment, which is based on Pigeon Hole theorem. Pigeon Hole theorem is that if there are m pigeon and *m+1* hole which can be occupied by just one pigeon, then, one of the holes has to be blanked at least. So, if *m+1* keys are selected as search key to guarantee *m* allowable errors, then, one of the search keys dont has errors at least. Figure 3 shows the worst case of error distribution. 5 errors distributed across the different keys. In this case, if we select any 6 keys of total keys, we can get a key which do not has errors at least.

Avoid duplicated Adjacency filtering at using multiple searching keys When we tried to use *e+1* keys as searching keys of Adjacency filtering, if the fragment is passed, then, it is possible that there are duplicated Adjacency filtering computations. For example, if the fragment is exactly matched to the Reference sequence at certain coordinate and we selected *e+1* keys as searching keys, then, each Adjacency filtering corresponding to *e+1* keys find out same coordinate as potentially matching coordinate. This means that *e+1* duplicated Adjacency operations are computed for finding out just one possible location. We can filter out these duplicated and redundant Adjacency operations by comparing the expected coordinate as the result of Adjacency filtering and the coordinates, already found as potentially matching coordinate before taking Adjacency filtering. We preserve the potentially matching coordinates, the result of previous Adjacency filtering at specific sized, 100ea in our implementation, array to filter out the redundant Adjacency filter. If it is matched, then, we simply cancel the reserved Adjacency filtering.

Imbalance of the number of key entry size In section 3.1 , Adjacency filtering drastically reduces the number of string comparison perform. However, Adjacency filtering also requires large computational resources for searching the expected coordinate in hash table to decide whether this location can be potentially matched to fragment or not. The imbalance of key entry size exacerbates the problem because we have to perform Adjacency filtering as much as key entry size. Figure 4 shows the distribution of key entry size within one chromosome. If we use the fragments which are located at A position, about 150,000 Adjacency filtering performs are required. The number of fragments located at A position is over 1000. Now, the dominant computation requirement changes from the string comparison perform to Adjacency Filtering perform.

Cheap Key Selection If we allow *e* errors, *e+1* searching keys have to be selected to maintain comprehensiveness. Each key has its own key entry size. To avoid large Adjacent Filtering computation, we can select the smallest *e+1* keys as searching keys. Hash table already stores the entry sizes corresponding to each key and based on this information, we can easily sort the key at the order of entry size and select cheapest *e+1* keys as searching keys. In this case, we can drastically reduce the number of Adjacency Filtering computations and increase the sequencing performance without degrading comprehensiveness. We call this as Cheap Key Selection. Figure 5 shows the operation of Cheap Key Selection. First of all, we divide the fragments as keys, i.e. if each fragment has 108ea base pairs and we use 12ea key length hash table, then we can divide the fragments as 9 keys. After that, the keys of the fragment can be sorted at the order of key entry size and cheapest *e+1* keys can be selected as searching keys. Figure 7 shows the efficiency of Cheap Key Selection. The blue colored diagram is the original distribution of Adjacency Filtering and the red colored diagram is the distribution after Cheap Key Selection. The number of the fragments, which requires large Adjacency Filtering perform is drastically reduced. Figure 6 shows that the total number of Adjacency Filtering perform is reduced to 6.4% of original total number of Adjacency Filtering at the case of selecting first *e+1* keys as searching keys without sorting based on the entry size.

The Cheap Key Selection also requires computational resources and the number of computations for sorting is proportional to log (the number of keys within each fragment). However, the number of Adjacency filtering is proportional to (the number of keys within fragment) * log (the key entry size). Usually, the fragments have 100 200 base pairs and the key length of Hash Table is about 10 to 12, So, the cost of Cheap Key Selection computation is quite smaller than the cost of Adjacency filtering.

4. GPU Implementation

We implemented the new algorithm FastHASH in CUDA as well. Please note that this is not a final implementation. Currently, we assign each graphic processor one fragment. For each fragment, we then spawn 32 threads as a warp. For each thread, they have 2 phases

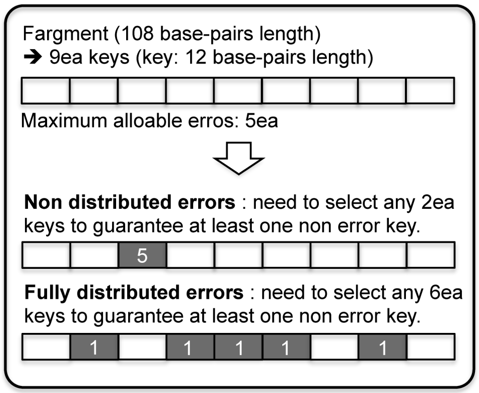
Fargment (108 base-pairs length)
➔ 9ea keys (key: 12 base-pairs length)
Maximum alloable erros: 5ea
Non distributed errors : need to select any 2ea keys to guarantee at least one non error key.
5
Fully distributed errors : need to select any 6ea keys to guarantee at least one non error key.
1
1
1
1
1

Figure 3. Error distribution & required number of keys

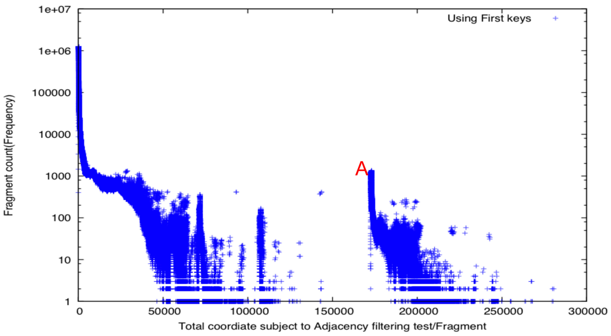

Figure 4. Adjacency Filtering Distribution

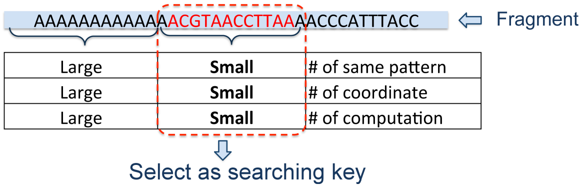

Figure 5. Cheap Key Selection Mechanism

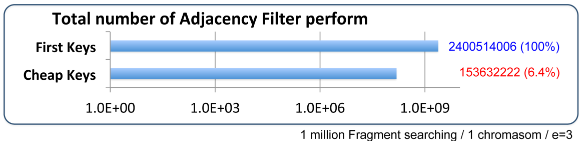

Figure 6. Cheap Key Selection Result

in calculation. Phase 1 is the adjacency filtering and phase 2 is edit-distance calculation. To keep as much thread in a warp as possible, we synchronize between 2 phases. The flow chart is shown in Figure 4. We have three main problems with this nave implementation. We will describe them below.

One problem we face in our CUDA implementation is that the edit distance calculation diverges a lot in control path. This is because the algorithm we implement is a lazy path filling algorithm* which tries to reduce work as much as possible. Such reduction introduced a huge amount of divergence, since each path may differ in ending point and they make the cheapest decisions based on previous execution result. Divergence hurt GPU drastically since all the threads within a warp have to always execute both paths.

Another problem with GPU is that for each fragment, the number of coordinates that passes through adjacency filtering differs a lot. For a 32 threads warp, to reach its maximum performance, we would like to have 32 threads working on edit distance calculations at the same time. As a result, in order to efficiently uses GPU, we would like each fragment has a multiple of 32 coordinates passes adjacency filtering, which, as we will show later in evaluation section, is not true for real workload. In fact, the real workload is a lot worse. For most of the time, the warp cannot be even filled up. One way to fix this problem, as we are still exploiting, is to bind multiple fragments to 1 processor, thus increasing the coordinates number that pass adjacency filtering and subject to edit-distance calculation. The third problem is that currently we are statically distributing workload. We assign all processor the same amount of fragments to align. Again, due to the intrinsic difference in edit-distance cost and adjacency filtering cost for each fragment, the total work and execution time for each processor varies. However, since we have to wait for the last processor to finish, the slowest processor will dominate the execution time. We may solve this problem by designing a smarter scheduler that dynamically redistributes fragments.

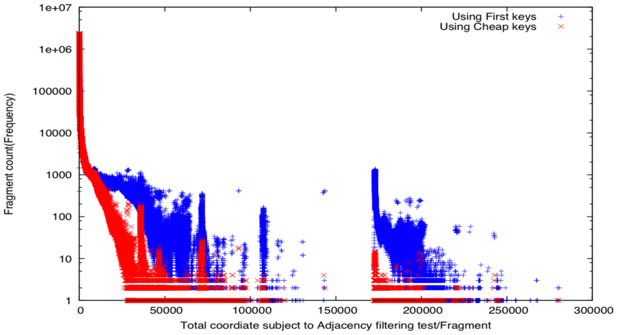

Figure 7. Efficiency of Cheap Key Selection

5. methodology

5.1. Hash table & Key size

The number of base pairs of whole DNA sequence is about 3.15billion. So, the number of coordinate corresponding to the whole DNA sequence is also the same with the number of base pairs of whole DNA sequence. About 12GBytes memory capacity is needed just for preserving the coordinates in hash table. To reduce the requirement of huge memory capacity to preserve the hash table, we divide the whole DNA sequence to 22 hash tables based on the chromosomes. The memory size to preserve the index of hash table is directly related to the key length, i.e. total 10.5MByte is need for preserving the index of hash table, corresponding to 10 base pairs length key and if we increase the number of key size to 15 base pair length, then, we need 16Gbyte just for preserving the index of hash table. Large table size also requires huge amount of time to transfer data from files to main memory. There is one more consideration point for choosing the key length as index. If we reduce the key length for index of hash table, then, the number of coordinates corresponding to each index is drastically increased. This means that it take long time to find out the coordinate for retrieve the reference sequences for sting comparison, also it requires a lot of time for Adjacency Filtering. The number of operations required for Adjacency Filtering is proportional to log (the number of coordinate corresponding to the index). The two factors, the requirement of memory capacity for hash table and the searching time took for Adjacency Filtering, are tradeoff. In this work, we choose 12 key lengths as index to balance these two factors. The key length of index can be changeable to optimize hash table for the specific purpose or the environmental change, i.e. the advent of huge memory size at low cost.

5.2. Input fragment set

The input fragments are selected from the substrings of the first chromosome of Reference sequence. After that, the selected substrings are shuffled with random for eliminated the similarity between adjacency fragments and reduce the effect of caching caused by the similarity of these adjacency fragments. These randomization methods make the input set to be more near the real input fragments. Total 1million fragments are selected as input fragments set by using these methods. The size of input fragment is 108ea base pair length.

5.3. Hardware environment

For the evaluation of CPU version, we use Intel Sandybridge i7 with 16GB main memory. For the GPU version, Nvidia Tesla C2070 with 6GB local memory (GDDR5) is used. The block size of GPU version is 280ea and the number of thread per block is 1000ea. Each block operates for just one input fragment and the threads corresponding to each block are allocated for the test of each coordinates corresponding to the fragment. For example, 280ea input fragments are fetched at same time and they are shipped to GPU memory. In GPU, there are 14ea computational cores and each core can perform one block at one time. In each GPU core, 1000ea threads can be generated for evaluate 1000ea coordinates. The 32ea threads can be operated simultaneously. Table 1 shows that the general specification of Nvidia Tesla C2070.

	Specification
Clock rate	1147000
Global memory	5.6GByte
Constant memory	65.5KByte
Multiprocessor	14ea
Shared memory / mp	49.2KByte
Register / mp	32.7KByte
Max threads / block	1024
Max threads dimensions	1024 / 1024 / 64
Max grid dimensions	65K / 65K / 65K

Table 1. Nvidia Tesla C2070 Specification.

6. Evaluation

6.1. The goal and quality of sequencing process

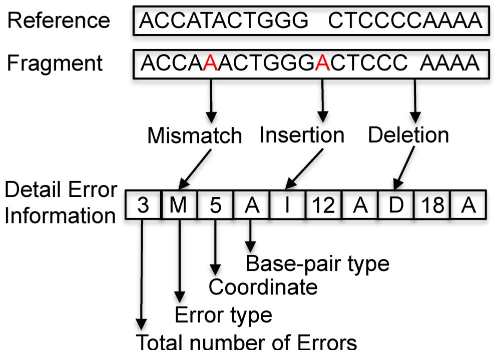

Figure 8. The data structure of detail error

The final goal of sequencing process is to find out the potential matching coordinates, the number of potential error corresponding to each potential matching coordinates and the detail information of the difference between input fragment and subsequences of that coordinates. The potential matching coordinates are starting coordinates of subsequences whose edit-distance from the corresponding fragment is under the saturation value (maximum edit distance), which is decided by the user. The detail information of the difference involve the type of differences (one of insertion, deletion and mismatch), the location of each differences and the sort of base pair corresponding to the location of difference. Figure 8 shows the example of the detail difference information. The quality of sequencing has two factors, time and comprehensiveness. The matching time means how long does it takes to perform and the comprehensiveness means that how many coordinate can be determined by matching process. So, it can be explained at the concept of coverage, like how many portion of potential matching points can be found by the matching process.

6.2. CPU evaluation and Result

First of all, we have to retrieve the hash table to memory for sequencing the input fragment set to Reference sequence. On section 5.1, we extract 22 hash tables corresponding to each chromosome fro Reference sequence. The matching sequence for the every input fragments is performed for one of the hash table first. After finishing the matching sequence for the hash table, then the next hash table is retrieved to the main memory and the matching sequence is performed again. The reason why we tried to match all the input fragments to one hash table first is that the required

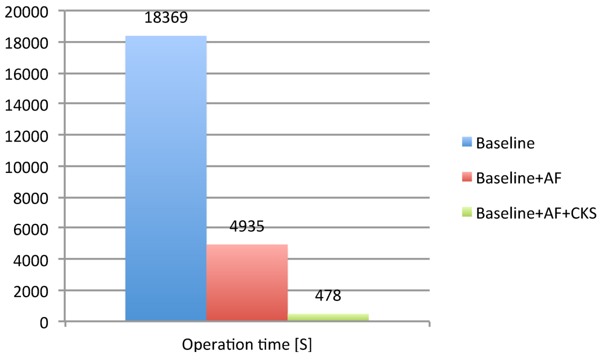

Figure 9. The total sequencing time of CPU implementations

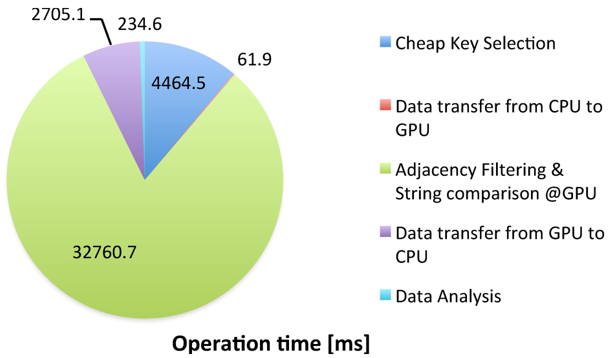

Figure 10. The total sequencing time of CPU implementations

time for loading hash table takes long time. So, the duplication of hast table loading process needs to be prevented to improve performance. There are three comparison methods of sequencing algorithm. The base line of CPU implementation is no filtering case and its algorithm is almost equal to the algorithm of mrFAST. After fetching the fragment, it is divided as several keys whose size is same with hash table keys length. If *e* error is allowed, first *e+1* divided fragment sequences are selected as the keys and it tries to find out the corresponding coordinate with the each key from the hash table. The subsequence of Reference sequence, which is located at the coordinate corresponding to the key, is retrieved from the Reference sequence. These two sequences are compared with each other by string comparison operation. If the edit-distance between these two sequences is lower than the maximum allowable edit-distance, then this location is determined as the potential matching location and the detail information of difference is informed. In this base line, there is no filtering method, so the string comparison operation is performed as many as the summation of all keys corresponding coordinate size. Second method uses the Adjacency Filtering. Before taking a string comparison, the Adjacency Filtering is performed to filter out the obviously not matched coordinates. After Adjacency Filtering, the string comparison is performed only when it is passed the Adjacency Filtering. The third method is similar with the second method. The only difference is that the cheapest *e+1* key is selected as the searching keys. Figure 9 shows the result of the three versions of CPU implementation. The case using only the Adjacency Filtering provides 3.7-fold speed up over the base line. The case using both of the Adjacency Filtering and Cheap Key Selection provides 38.4-fold speedup.

6.3. GPU evaluation and Result

Figure 10 shows the time consumptions corresponding to each stage of sequencing. In the stage of Cheap Key Selection, the input fragment (108 base-pairs length) is divided to 9ea 12 base-pairs length keys and sorted these keys based on the number of coordinates corresponding to the keys. It takes about 11% of total sequencing time. In the stage of input data transfer, the sorted fragments are shipped to the GPU. After that, GPU operated Adjacency filtering and string comparison and it takes about 81.4% of total sequencing time. The output data transfer time is about 6.7% of total sequencing time. The reason why output data transfer time is quite larger than the input transfer time, is that output data involves the potential locations, the number of edit-distance and the detailed information of difference between fragments and Reference.

7. Analysis

As it is shown in Result section, Adjacency filtering reduces the number of edit-distance performed a lot. However, as the adjacency filtering will always test if N e segments find corresponding locations, the effectiveness of adjacency filtering is related to the user-set error tolerance number e. As e increases, the effectiveness of adjacency filtering decreases, since the required number of matching segments is reduced. To maintain the adjacency filtering effectiveness, one solution will be using shorter key for hash table thus cut the fragments into shorter segments and increasing the segments count. For example, if the fragment is 108 base-pairs in length and each segment is 12 base-pair in length so we cut the fragment into 9 segments, for a user set error tolerance number e=5, instead of requiring N-e = 9-5 = 4 segments find corresponding adjacent locations, we now reduce the key length to 9 which will divide the fragment into 12 segments, and we will require N-e = 12-5 = 7 segments finding corresponding adjacent locations. We have simulated 9 case for 1 chromosome and it turns out when e=5, setting key length to 9 does increase the effectiveness of adjacency filtering where it is filtering out more not matching locations. However, we do see

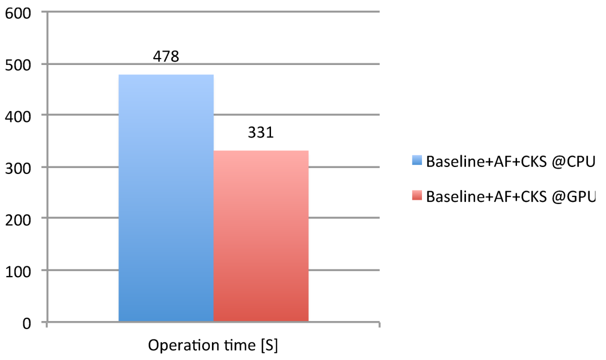

Figure 11. The total sequencing time of CPU and GPU implementations

a longer execution time. The reason is that although the adjacency filtering effectiveness is enhanced, the cost of adjacency filtering is increased. Since now the keys are shorter in hash table and the entries are fewer while the total number of coordinates stays the same as before, the coordinate list for each hash table entry is increased. As a result, adjacency filtering will need to reverse more coordinates. To make it even worse, now there are more segments and thus more searching for corresponding adjacent coordinates. Last but not the least, this increase the chance of encountering popular keys with the same reason we described in Algorithm section, as now we are having less entries but longer coordinate list. In all, this reduces the execution time a lot. In figure 7, we show when changing key length from 12 to 9. How many coordinate are subjects to adjacency filtering compared to before. In figure 8, we show the increased effectiveness of adjacency filtering since less coordinates will be subject to edit-distance calculation and figure 9 shows the increased execution time.

For cheap key selection, we face the same problem. Since we are picking the cheapest N-e+1 keys. As e increases, the keys we pick will become less *cheap*. When e = N-1, the program will just behave as if without cheap key selection, since it is picking all the keys anyhow. We can increase the chance of picking relatively cheap keys by the same trick as above: smaller key length and more segments. However, by also same with the reason listed above, increasing the key number does not necessarily provides speed up. Although we increase the chance picking relatively cheap keys, since the coordinate list is longer now, the now *cheap* keys coordinate list are actually longer than before. We modeled key length 9 case, and the result is the shown in figure 7 as well.

In reality, normally error tolerance e is set to 5% of the fragment sequence length. For a 108 read, e will normally be 5, by which FashHASH still provides generally considerable speed up. However, as e increases, the performance improvement will diminish, because of the reason stated above.

References

[1] Applied Biosystems. The Applied Biosystems SOLiD System. `http://http://marketing.appliedbiosystems.com/images/Product/Solid_Knowledge/flash/102207/solid.html/`.

[2] D. R. Bentle. Whole-genome re-sequencing. Technical report, Current Opinion in Genetics & Development, 2006.

[3] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[4] P. e. a. Campbell. Identification of somatically acquired rearrangements in cancer using genome-wide massively parallel paired-end sequencing. Technical report, Nat. Genet., 2008.

[5] D. e. a. Chiang. High-resolution mapping of copy-number alterations with massively parallel sequencing. Technical report, Nature Methods, 2009.

[6] A. et al. mrsfast: a cache-oblivious algorithm for short-read mapping. Technical report, Natura Methods, 2010.

[7] L. et al. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 2009.

[8] R. et al. Shrimp: Accurate mapping of short color-space reads. *PLoS*, 2009.

[9] R. et al. Comparative analysis of algorithms for next-generation sequencing read alignment. Technical report, Bioinformatics, 2011.

[10] P. Ferragina, G. Manzini, V. Mkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3:2007, 2007.

[11] L. H. and D. R. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 2009.

[12] J. Shendure and H. Ji. Next-generation dna sequencing. Technical report, Nature Biotechnology, 2008.

[13] J. e. a. Wang. The diploid genome sequence of an asian individual. Technical report, Nature, 2008.

[14] D. e. a. Wheeler. The complete genome of an individual by massively parallel dna sequencing. Technical report, Nature, 2008.