Efficient GPU Acceleration for Computing Maximal Exact Matches in Long DNA Reads

Ahmed, Nauman; Bertels, Koen; Al-Ars, Zaid

DOI
10.1145/3386052.3386066

Publication date
2020

Document Version
Accepted author manuscript

Published in
ICBBB 2020

Efficient GPU Acceleration for Computing Maximal Exact Matches in Long DNA Reads

Nauman Ahmed
Delft University of Technology
Delft, Netherlands
n.ahmed@tudelft.nl

Koen Bertels
Delft University of Technology
Delft, Netherlands
k.l.m.bertels@tudelft.nl

Zaid Al-Ars
Delft University of Technology
Delft, Netherlands
z.al-ars@tudelft.nl

ABSTRACT

The seeding heuristic is widely used in many DNA analysis applications to speed up the analysis time. In many applications, seeding takes a substantial amount of the total execution time. In this paper, we present an efficient GPU implementation for computing maximal exact matching (MEM) seeds in long DNA reads. We applied various optimizations to reduce the number of GPU global memory accesses and to avoid redundant computation. Our implementation also extracts maximum parallelism from the MEM computation tasks. We tested our implementation using data from the state-of-the-art third generation Pacbio DNA sequencers, which produces DNA reads that are tens of kilobases long. Our implementation is up to 9x faster for computing MEM seeds as compared to the fastest CPU implementation running on a server-grade machine with 24 threads. Computing suffix array intervals (first part of MEM computation) is up to 3x faster whereas calculating the location of the match (second part) is up to 9x faster. The implementation is publicly available at https://github.com/nahmedraja/GPUseed

KEYWORDS

DNA analysis, GPU, seeding, maximal exact matches

1 INTRODUCTION

Current DNA analysis programs have to process massive amounts of data. In many applications this data is in the form of *DNA reads*. A DNA string is a single read or an assembly of many reads. It is made up of only four characters: 'a', 'c', 't' and 'g'. These four characters represent the four types of nucleotide bases in the DNA and are also known as *base pairs* (bp). Many of the DNA analysis algorithms require to solve an approximate string matching problem. Therefore if the number of the DNA strings to be matched is large or the strings are long, the direct application of Smith-Waterman [1] or similar algorithms is too slow for practical purposes. To overcome this problem, BLAST [2] pioneered the approach of applying a *seeding* heuristic. The reason behind applying the seeding heuristic is the observation that for a good match to be found between two strings, these strings must share a highly matching substring. Therefore, to match two strings, first a common substring should be found. This common substring is known as a *seed*. An approximate match between the strings is then found by applying Smith-Waterman around the seed. This two-step method is known as seed-and-extend.

Computing a seed refers to finding the pattern of the common substring as well as its location in the two strings. In a typical DNA analysis application, millions of independent DNA strings have to be processed, each of which may contain many seeds. Therefore, computing seeds is a highly parallel problem, that can be accelerated on massively parallel Graphical Processing Units (GPUs) devices. In this paper, we present the fastest GPU implementation for computing *maximal exact matching* (MEM) seeds on GPUs in state-of-the-art third generation long DNA reads.

This paper is organized as follows. Section 2 discusses background information. Section 3 describes previous research work. Section 4 describes our GPU implementation. Section 5 presents the experimental results. Section 6 concludes the paper.

2 BACKGROUND

As described in the introduction, seeding is used to speed up the process of finding an approximate match between two DNA strings. More formally, let one string be called as text T and the other be called pattern P. In practice, T is one long DNA string assembled from many DNA reads. To find an approximate match between T and P, we first need to find substrings of P that are exactly matching at one or more locations in T. A *maximal exact match* (MEM) between two strings, T and P, is an exactly matching substring of T and P that cannot be further extended in either direction without incurring a mismatch. We call this *definition 1* of MEM. The concept of maximal exact matches was first proposed in [3]. If either T or P is a long string then the number of MEMs could be quite large. Furthermore, a MEM computed using definition 1 can be a substring of another MEM causing repetition. Therefore an alternative definition (*definition 2*) avoids the repetition. According to definition 2, a substring of P is a MEM if it is located at n positions in T, i.e. matches with n identical substrings of T, and cannot be further extended in either direction at all n locations. In this paper we will use definition 2 for a MEM.

2.1 The FM-index

The algorithm for computing the seeds depends upon the underlying *index*. The index is a pre-built data structure that is used to compute seeds. For example, to find substrings of pattern P that exactly match in text T, an index of T is built. In DNA analysis applications, mainly two types of indexes are used: i) Hash tables ii) Indexes based on suffix/prefix trie. *FM-index* [4] is a popular index based on the suffix trie. It is widely used in many DNA analysis applications due to its speed and small memory footprint. In our GPU implementation, we have used the seed searching algorithm based on the FM-index. FM-index is a set of three arrays: i) Count array C ii) Burrows-Wheeler transform array BWT and iii) Suffix position array SP. The C array has only four elements, one for each DNA base. The BWT array holds the Burrows-Wheeler transform of T. The SP holds the starting position of the suffixes of T. Usually, to save RAM, SP is stored in compressed form and the starting

Algorithm 1: Computing Suffix array intervals of all the MEMs between P and T

Input: Pattern P and minimum required MEM length min_mem_len
Output: Array M containing all the MEMs in P

```
1  Function MemSAInterval(P, min_mem_len) begin
2      Initialize M as an empty array
3      for j ← |P| − 1 down to min_mem_len − 1 do
4          [l, u] ← [0, |T| − 1]
5          q ← j
6          while q >= 0 do
7              prev_l ← l
8              prev_u ← u
9              l ← C[P[q]] + Occ(P[q], l − 1) + 1
10             u ← C[P[q]] + Occ(P[q], u)
11             if l > u then
12                 break
13             q ← q − 1
14         // q = −1
15         if l ≤ u and j − (q + 1) + 1 ≥ min_mem_len then
16             start ← q + 1
17             end ← j
18             Append ([l, u], start, end) to M
19         // otherwise
20         else if j − q + 1 ≥ min_mem_len then
21             start ← q + 1
22             end ← j
23             Append ([prev_l, prev_u], start, end) to M
```

position of only those suffixes is stored for which SP array index is a multiple of a certain number, known as *compression ratio* in this paper. [4] contains a detailed discussion on FM-index.

2.2 Computing MEM seeds using FM-index

As described before, computing a seed means that we attempt to find a common substring between T and P and find its location in T and P. Assume $P[i:j]$ is a substring of P. i and j are the starting and ending positions of $P[i:j]$ in P. $|.|$ denotes the length of a string. Seed computation using the FM-index is completed in two steps: a) Computing suffix array intervals of the seed, and b) Locating the seed in T using suffix array intervals.

2.2.1 Computing suffix array intervals. Suffix array interval $[l(P[i:j]), u(P[i:j])]$ of $P[i:j]$ is defined as:

$$l(P[i:j]) = \min\{k : P[i:j] \text{ is the prefix of } SA[k]\}$$
$$u(P[i:j]) = \max\{k : P[i:j] \text{ is the prefix of } SA[k]\}$$

where suffix array SA contains the lexicographically sorted suffixes of the text T. Suffix array interval of $P[i:j]$ can be computed using FM-index with *backward search*. In backward search we start with an empty string. The l and u of the empty string are defined as 0 and $|T|-1$, respectively. We then add bases to the empty string from the end of $P[i:j]$, one base at a time, in the backward direction so that the string grows as $P[j:j] \rightarrow P[j-1:j] \rightarrow P[j-2,j] \cdots \rightarrow P[i:j]$. After adding every base the suffix array interval of the new string is calculated as

$$l(P[x:j]) = C[P[x]] + Occ(P[x], l(P[x+1:j]) - 1) + 1 \quad (1)$$
$$u(P[x:j]) = C[P[x]] + Occ(P[x], u(P[x+1:j])) \quad (2)$$

where $Occ(b, y)$ is the number of occurrences of base b in the BWT array from 0 to y. If $l(P[x:j]) \leq u(P[x:j])$, then $P[x:j]$ does exist in T and occurs at $u(P[x:j]) - l(P[x:j]) + 1$ locations in T.

Algorithm 2: Computing the starting position for a given suffix array index

Input: suffix array index of the seed sa_idx
Output: starting position of the seed in the text T

```
1  Function LocateSeed(sa_idx) begin
2      itr ← 0
3      i ← sa_idx
4      while i%r ≠ 0 do
5          i ← C[BWT[i]] + Occ(BWT[i], i − 1)
6          itr ← itr + 1
7      return SP[i] + itr
```

Algorithm 1 uses the recurrence Equations 1 and 2 to compute the suffix array intervals of all the MEMs between P and T. The algorithm returns an array M containing the MEMs in the form of tuples $([l,u], start, end)$, where $[l,u]$ is the suffix array interval of the MEM; $start$ is the starting position of the MEM in P and end is the ending position of the MEM . MEM computation starts from the base at index q and builds the MEM string in the backward direction by adding bases from P until the base at the $0th$ index of P is reached.

2.2.2 Locating the seed. Locating a seed refers to computing the start position of seed in T. Algorithm 2 shows how the FM-index is used to compute the starting position of seeds with a compressed suffix position array SP having a compression ratio of r. The suffix array interval of a seed contains $u - l + 1$ suffix array indexes. The `LOCATESEED` function accepts a suffix array index (sa_idx) of the seed and computes the corresponding seed starting position in T. Hence, the `LOCATESEED` function is called for $sa\_idx = l, l + 1 \ldots, u - 1, u$ to find all the locations of the seed in T.

2.3 Graphical processing units

In heterogeneous computing era, Graphical Processing Units (GPUs) are used as accelerators for high performance applications due to there massively parallel architecture. In the following, we will briefly describe the architecture and programming of NVIDIA GPUs

In NVIDIA GPUs, there are numerous streaming multiprocessors (SMs). Each SM has many cores known as streaming processors (SPs). An SP is the basic computational unit that executes a GPU thread. GPU also has its own DRAM known as *global memory*. GPU threads are grouped into *blocks*. Each block contains many GPU threads. The threads in a block are assigned to the same SM. The number of blocks and the number of GPU threads in a block are configured by the programmer. All the GPU threads can communicate with each other via global memory. However, the threads in a block can also communicate through a fast *shared memory*. In a block, there is a set of 32 threads known as a *warp* that share the program counter. The threads in a warp can communicate with the help of *shuffle* instructions.

The programming language for NVIDIA GPUs is known as *CUDA* which is an extension of C/C++. The GPU programmer writes a *kernel* which executes on the GPU. The GPU is attached to a *host* CPU which initiates all the tasks related to the GPU. The data to be processed by the kernel is copied from the host memory to the global memory of the GPU. The CPU then launches the kernel. Once the kernel is finished the results are copied from the global memory back into the host CPU memory.

3 PREVIOUS RESEARCH WORK

Maximal exact match seeds are computed in a variety of bioinformatics applications which include: BWA-MEM [5] and CUSHAW2 [6] DNA read mappers; Jabba [7], a DNA sequencing error correction tool; and MUMmer [3] a whole genome alignment application. Therefore, optimizing or accelerating the computation of the maximal exact matches could be beneficial for a large number of DNA analysis programs. Some seeding algorithms, like slaMEM [8], extend the FM-index for faster computation of MEMs, while others, like essaMEM [9], propose other types of indexes to speed up the computation. CUSHAW2-GPU [10] is the GPU implementation of the CUSHAW2 read mapper. It computes the MEMs on the GPU. But it is designed only for short reads with a maximum allowed read length of only 320 bases. Moreover, it stores the suffix array intervals of the MEMs in an intermediate file which is subsequently loaded for locating the MEM seeds on the reference genome. This makes the computation extremely slow. In [11] the authors present the computation of exact matches using suffix trees. Suffix trees are large data structures that consume a huge amount of memory. The suffix tree used in the paper would take 152 gigabytes of memory for the human reference genome. Hence, the approach is impractical for large genomes. GPUmem [12] is a GPU implementation for computing maximal exact matches between only two very long DNA sequences, e.g. between chromosome 1 and chromosome 2 of the human genome.

NVBIO [13] is an open source library developed by NVIDIA. It contains the functions for computing the maximal exact matching seeds on GPU using the FM-index. But it can only be used to find maximal exact matching seeds for short DNA reads. For long DNA reads, NVBIO terminates due to illegal memory access error. This may be due to the sizes of some data structures that are not sufficient for long DNA reads. Moreover, NVBIO uses bidirectional BWT of [14] which limits the amount of parallelism (See Section 4.3, first paragraph). This is especially true for long DNA reads since less number of long DNA reads can be loaded in the GPU memory as compared to short DNA reads for MEM computation.

Computing MEM seeds is a highly parallel process. The seeds for all the DNA reads can be computed in parallel. Even for a single read the iterations of the `for` loop in Algorithm 1 are independent. Moreover, each MEM occurs at $l - u + 1$ places in T and all these locations can be calculated in parallel using Algorithm 2. Hence, the computation of maximal exact matching seeds is well suited for parallel processing. Maximal exact matching seeds are widely used for DNA analysis but in the past, there were limited efforts to parallelize their computation. State of the art third generation sequencing platforms producing long DNA reads with lengths up to hundreds of kilobases. Therefore, in this paper, we present a high performance GPU implementation for computing maximal exact matching seeds in long DNA reads. The contributions of the paper are as follows:

- We present the fastest GPU implementation for computing the maximal exact matches for state-of-the-art third generation long sequencing reads for DNA analysis.
- We present a unique optimization of early detection of redundant MEMs by using CUDA *warp shuffle* instruction
- Experiments show that our implementation provides 7-9x speedup over the fastest CPU implementation for third generation Pacbio DNA reads

4 GPU IMPLEMENTATION

Our GPU implementation exploits the massive parallelism in seed computation. In many situations, there is a large number of patterns that need to be matched with the text T. Moreover, several seeds for each pattern P need to be computed. Therefore, in total, a large number of seeds can be computed in parallel. The GPU implementation contains several stages for computing MEM seeds. Figure 1 shows the different computational stages of the implementation. The array shown after each stage is the output of the stage. Each stage launches one or more GPU kernels.

4.1 The index

To compute the seeds, a pre-built index is loaded in the GPU memory. The index consists of BWT array, count array C, compressed suffix array SP and a pre-calculated suffix interval array. The use of a pre-calculated suffix interval array is described in Section 4.3. As described in Section 2.2, the seed search algorithm using FM-index needs to compute $Occ(b, y)$, which is the number of occurrences of base b in the BWT array from 0 to y. In the case of a large BWT array counting the number of occurrences become very slow. Therefore, in practice, the BWT array is divided into *bins*. A *checkpoint* exists between two consecutive bins that contain the number of occurrences of all the four types of bases till that checkpoint. To find $Occ(b, y)$, first the *bin* containing the index y is found. Then the bases of type 'b' in the bin till index 'y' are counted and the count is added to the preceding checkpoint value of the base b. The checkpoints occur after a fixed number of BWT bases. This number is known as bin_size. In the implementation, each BWT array entry is a 32-bit integer. Since each base is encoded in 2 bits, the bin_size must be a multiple of 16. We found that $bin\_size = 64$ is the most suitable choice as it allows to load the checkpoint and the bin using only 2 `uint4` CUDA vector load instructions. CUDA *popcount* instructions are used to count the number of bases in a bin. We also tested other bin sizes. Smaller bin sizes help to decrease the number of popcount instructions but cannot decrease the number of load instructions as the minimum bin size is 16. Moreover, smaller bin sizes can increase the number of memory accesses as described in the first optimization of Section 4.3.

4.2 Stage-0: Pre-processing

GPU accepts input data from the CPU and returns the results. The input data is the concatenated patterns for which the seeds are to be computed on the GPU along with the length of each pattern. The user also passes the pointer to the pre-built FM-index arrays. The index is constructed using the `index` command in BWA version 0.5.9 [15]. The index is copied to the GPU global memory. In the rest of the paper, we will assume that a pre-built FM-index exists in the GPU global memory. The implementation preprocesses the input to generate 4 arrays. All the arrays are computed on the GPU: 1) a 32-bit integer array which contains packed patterns. The bases of the input patterns are converted from ASCII to 4-bit representation. 4-bit representation is required to accommodate the fifth type of base

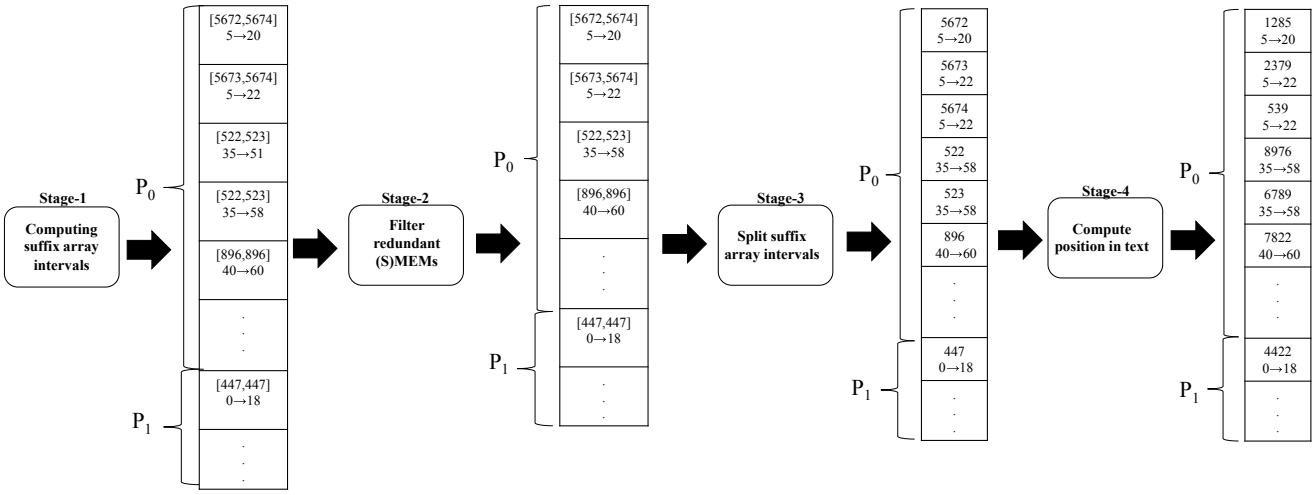

Figure 1: Different computational stages in the GPU implementation

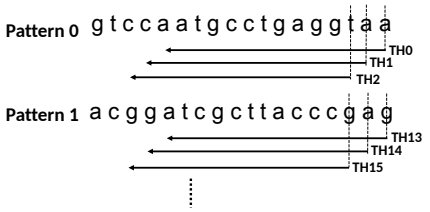

Figure 2: An example of GPU thread assignment for MEM seed computation on GPU. Minimum required seed length is 6

'n', known as an ambiguous base. 4 bits instead of 3 bits are chosen for the ease of post-processing. 3-bit representation allows to pack 10 bases in an integer instead of 8 and hence provide a significant advantage over 4-bit representation. The packing is performed on the GPU using the same method as described in [16]. 2) an integer array containing the starting index of the patterns in the input. 3) an integer array that assigns a pattern to a thread. Each GPU thread computes the suffix array interval for one MEM 4) an integer array which assigns a MEM within a pattern to a thread. There is a large variation in the length of the long DNA reads produced by the third generation DNA sequencers. Therefore, the computation of the 3rd and 4th array is essential for the efficient utilization of the GPU resources. Since all the output arrays of the pre-processing stage are computed on GPU, the overhead of the pre-processing is negligible. We found that the total time spent in prepossessing is less than 1% of the total execution time in all the experiments.

4.3 Stage-1: Finding suffix array intervals

The software optimized algorithms for computing MEMs are different from the one shown in Algorithm 1. For example, the bidirectional-BWT proposed in [14] allows adding bases both in the forward and backward direction. Also in [17] and in BWA-MEM [5] a bidirectional FM-index (called the FMD-index) is implemented. In such algorithms, the MEMs covering an index v of pattern P are computed by first adding the bases in the forward direction and after adding every base the corresponding suffix array interval is stored. Hence, the stored suffix array intervals are for the strings: $P[v]$, $P[v]P[v+1]$, $P[v]P[v+1]P[v+2]$, ... Then each of these strings is extended in the backward direction to compute all the MEMs covering base at index v. The bases added in the forward direction are common to all the MEMs and hence added only once. This reduces the accesses to the FM-index. One disadvantage of such an approach is that the forward addition of bases has to be sequential and cannot be parallelized. Therefore, in our GPU implementation, we will use Algorithm 1 as it allows all the MEMs in pattern to be computed in parallel exploiting the massive parallelism of GPUs.

The main GPU kernel in this stage is used to compute the suffix array intervals of MEMs. One MEM is assigned to a GPU thread. Figure 2 shows an example of the GPU thread assignment. *THx* are the threads, where x is the thread number. Each GPU thread starts from a different base of the pattern as shown in Figure 2 and then extends it in the backward direction. The kernel is similar to Algorithm 1, but each iteration of the `for` loop is computed by a separate GPU thread. We further applied two optimizations which are explained below:

4.3.1 First optimization: pre-calculated suffix array intervals. As a part of the index building, we pre-calculate the suffix array intervals of all the possible $4^{PRE\_CALC\_LEN}$ sequences of length PRE_CALC_LEN. Therefore, the suffix array interval of the last PRE_CALC_LEN bases of the MEM is already known. The GPU thread first loads the pre-calculated suffix array intervals of the last PRE_CALC_LEN and then extend the MEM in the backward direction. Using pre-calculated suffix array intervals has two advantages

(1) The pre-calculated suffix intervals reduce the number of backward search steps required for the computation of the suffix array interval of a MEM.
(2) As shown in Algorithm 1 the calculation of a suffix array interval $[l, u]$ requires two accesses to the BWT array to

compute the *Occ*. At the start of the backward search, these accesses are in different bins of the *BWT* array, but as the backward search proceeds the difference between l and u decreases [18] and at some point may become less than the bin size. From here only one *BWT* array access is required to compute the suffix array interval. Using the pre-calculated suffix array intervals may allow us to skip that initial phase of two *BWT* array accesses.

We found that $PRE\_CALC\_LEN = 13$, requiring 512 megabytes of GPU memory, provides a good speed-memory tradeoff. In case the minimum required MEM length is less than 13, the value of *PRE_CALC_LEN* is reduced.

As described previously, the MEM finding algorithm optimized for CPUs use the bidirectional index and the algorithm is different for the one used in our GPU implementation (Algorithm 1). The downside of using such an index is that the optimization of pre-calculated suffix array intervals cannot be applied. Hence, this optimization is unique in the sense that the MEM finding algorithm of our GPU implementation allows to pre-calculated suffix array intervals to speed up the MEM computation.

4.3.2 Second optimization: Early detection of redundant MEMs. Two overlapping MEMs may have the same suffix intervals and hence, the smaller one is redundant. For example in Figure 2, $TH1$ and $TH2$ may have the same suffix intervals at the same index of Pattern 0 and will be backward extended till the same base of the pattern. In this case, the MEM found by $TH2$ is redundant. The kernel for finding the suffix array intervals of the MEMs tries to detect such redundant MEMs. A GPU thread keeps record of the previously computed suffix array interval sizes ($u - l + 1$) in the *prev_intv_size* array during the backward search. A MEM computed by a GPU thread is redundant if a value in its *prev_intv_size* array is same as the current interval size of a thread with lesser thread ID, which is in the same warp and is also working on the same pattern. If the MEM assigned to GPU is found to be redundant by the early detection mechanism, the thread exits. To access the values of the current interval size of the other threads, we use CUDA *warp shuffle* instruction, which allows the rapid exchange of variables between threads in the same warp without involving shared memory.

Finally, each thread writes the suffix array interval and start and end position of the MEMs in the output array as shown in Figure 1. Each entry in the output array of Stage-1 contains two values. The top value is the suffix array interval and the bottom value is $start \rightarrow end$, where *start* and *end* are the starting and ending positions of the MEM in the pattern, respectively. Some entries will be NULL. Note that the values in the output array are in the ascending order with respect to the *start*. In Figure 1, the output of Stage-1 shows example values of suffix array intervals and $start \rightarrow end$.

4.4 Stage-2: Filtering redundant MEMs

A warp contains only 32 threads and a warp shuffle instruction only exchange variable within a warp. Therefore, the output of Stage-1 still contains some redundant MEMs. The GPU kernel of Stage-2 is used for this purpose. First the NULL entries in the output array of Stage-1 are eliminated. We used the `DeviceSelect` kernel from *CUB* GPU library [19] to eliminate these NULL entries and compact the output array of Stage-1. After compaction the ith threads is assigned the ith entry of the output array of Stage-1. The thread applies a test to know whether $start[i] == start[i-1]$ and $u[i] - l[i] + 1 == u[i-1] - l[i-1] + 1$. If both are true, the MEM corresponding ith entry is marked as redundant. Since each thread has to work on only one entry, the filtering stage is very fast. The output of this filtering kernel is once again compacted to remove the redundant MEMs using CUB `DeviceSelect` kernel. An example of the output of the filtering stage (Stage-2) after compaction is shown in Figure 1. It has the same format as Stage-1. Note that the third value in the output array of Stage-1 is filtered out.

4.5 Stage-3: Splitting suffix array intervals

This GPU kernel splits up the suffix interval to its constituent suffix array indexes. The splitting is done so that each GPU thread in the locate kernel (Section 4.6) computes only one position in the DNA text T corresponding to the suffix array index assigned to it. In the case of two overlapping MEMs with the same starting position, the suffix interval of the longer MEM is the subset of the shorter MEM. Therefore, the splitting up performed by this stage makes sure that a suffix array index does not appear twice and, hence two GPU threads in the locate kernel are never assigned the same suffix array index. This will reduce the number of locations to be computed by the locate kernel. The splitting process is very fast as each GPU thread is assigned only one suffix interval to split. This stage takes less than 1% of the total execution time in all the experiments. The output after this stage is shown in Figure 1. Each entry has two values. The top value is the suffix array index, while the bottom value is $start \rightarrow end$. Note that the suffix array index 5673 is present in both the first and second entries of the output of Stage-2, but it is not repeated in the output of Stage-3

4.6 Stage-4: Locating MEMs in text

This is the final stage and generates the locations of the MEM seeds in the DNA text T. Each GPU thread has to compute only one location using Algorithm 2. Figure 1 shows the example values of the seed locations in the text T.

5 EXPERIMENTAL RESULTS

For comparison purposes we considered the problem of computing MEMs between a set of DNA reads and human reference genome, UCSC hg19 (GRCh37 Genome Reference Consortium Human Reference 37 (GCA_000001405.1)). Hence, in this case the text T is the reference genome. MEMs on both the forward and reverse genome are computed. We used long DNA reads generated by state-of-the-art third generation whole genome sequencing Pacbio reads downloaded from [20]. The total number of reads are 25249 with the average read length of 7 kilobases and longest read is 35 kilobases.

Our GPU implementation is executed on NVIDIA Tesla K40c installed in a 2-socket Intel Xeon E5-2620 v3 CPU with two way hyper-threading (12 physical cores, 24 logical cores) and 32 gigabytes of RAM. The CPU implementation is executed on the same machine using 24 threads. We tried different block sizes i.e. the number of GPU threads per block. We found a block size of 128 to be a suitable choice.

The index is the same as described in Section 4.1 consisting of a *BWT* array (1.6 gigabytes) with *bin_size* = 64, count array (20 bytes), compressed suffix position array (1.8 gigabytes) with compression ratio of 7 and a pre-calculated suffix intervals array with *PRE_CALC_LEN* = 13 (512 megabytes). *PRE_CALC_LEN* = 12 for minimum MEM length of 12. Hence the total FM-index size is around 4 gigabytes. For the CPU implementation, the FM-index size is 4.8 gigabytes.

Figure 3 shows the time spent in different stages for computing MEMs in our GPU implementation with different minimum required MEM lengths. The "find intv" is Stage-1, which computes the suffix array intervals as described in Section 4.3. Stage-2 is shown in the figure as "filter". Stage-3 for splitting suffix array intervals and Stage-4 for locating the MEMs in the text T is shown as "locate" in the figure. The time spent in Stage-3 (splitting) is negligible as compared to Stage-4 (locating the MEMs) in the text T. "cudamemcpy" represents the time spend in transferring data from CPU to GPU and vice versa. We have not included the time to copy the FM-index and pre-calculated suffix array intervals to the GPU as it is done only once at the beginning of the program. The "cudamemcpy" time is mainly due to copying the final output array containing the positions of the MEMs in the text T and pattern from GPU to CPU. Time spend in copying the pattern (reads) sequences and their lengths and offsets take negligible time. The figure shows that the time for Stage-1 (finding intervals) remains constant (around 6 seconds) irrespective of the minimum MEM length. Stage-2 (filtering of the intervals) takes negligible time for all minimum lengths and is around 1% of the total execution time. The "locate" (Stage-3 and Stage-4) time is small for longer minimum MEM lengths but becomes dominant for smaller values of minimum MEM lengths. For a minimum MEM length of 12 "locate" is nearly 6 times higher than "find intv". This happens because of the number of MEMs increase with decreasing minimum lengths. For the same reason, time spent in memory transfers between CPU and GPU is small for longer minimum lengths but becomes significant for shorter values of minimum MEM lengths.

Figure 4 shows the comparison of time spent in different stages of the MEM computation in the CPU implementation and our GPU implementation. The CPU implementation actually has only two stages: finding the MEM suffix array intervals ("find intv") and locating the MEMs on the text T ("locate"). There is no separate filtering stage as it is performed during interval computation. The "cudamemcpy" in CPU implementation is zero. The figure shows that the main reason for the speedup of GPU over CPU is due to a much faster "locate" stage on GPU as compared to CPU. The "locate" on GPU is 7x-9x faster than CPU. The "find intv" stage of the GPU implementation is around 6 seconds for a minimum MEM length of 14-20. For a minimum MEM length of 12, it slightly increases and becomes 7 seconds. This happens because the *PRE_CALC_LEN* is decreased from 13 to 12. Overall, "find intv" is around 2-3x faster on GPU.

Figure 5 shows a comparison of our total GPU execution time to CPU time. Since 24 threads are used to execute the CPU implementation, the time spent in "find intv" and "locate" stage of the CPU implementation shown in Figure 4 is computed by taking the *average* across the stage time for all threads. Therefore, the sum of the time spent in the CPU stages do not add up to the total CPU execution time. For MEMs, the speedup of GPU over CPU varies from 7x-9x. The maximum speedup is achieved around the minimum MEM lengths of 16-18.

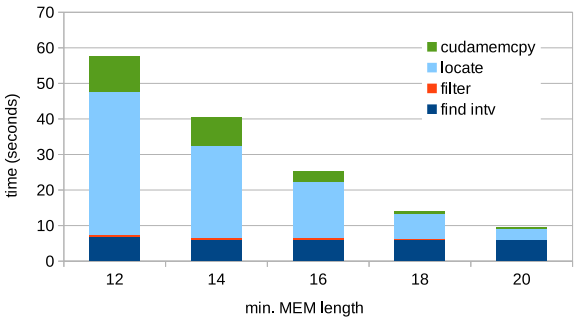

Figure 3: Time spent in different stages for computing MEMs on GPU for Pacbio reads

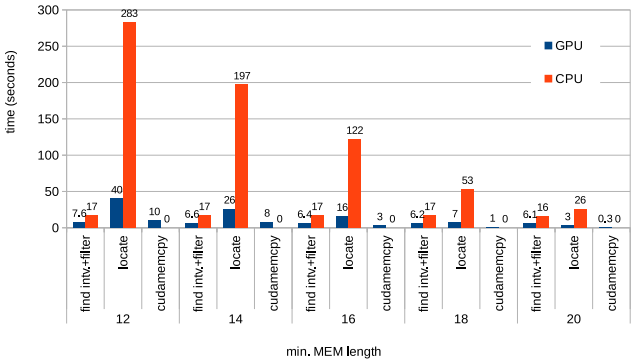

Figure 4: Comparison of time spent in different stages to compute MEMs on CPU and GPU for Pacbio reads. The CPU implementation is executed with 24 threads.

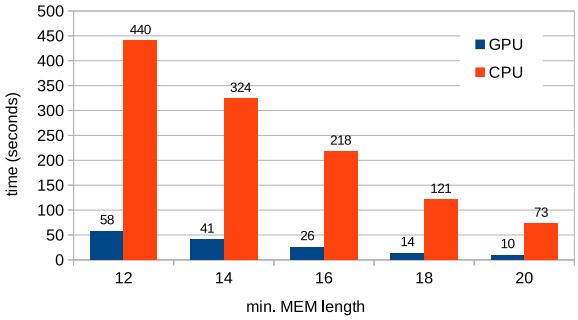

Figure 5: Comparison of total execution time to compute MEMs on CPU and GPU for Pacbio reads. The CPU implementation is executed with 24 threads.

6 CONCLUSIONS

In this paper, we presented a GPU acceleration of computing MEM seeds on GPU for state-of-the-art long DNA reads. The implementation computes the MEM seeds using the FM-index. Optimizations were applied to reduce the GPU memory accesses and to reduce redundant computation. We also extract maximum parallelism from the MEM computation task. Our GPU implementation is the fastest as compared to other available tools for computing MEMs in long DNA reads. Experiments were performed using the latest Pacbio DNA sequencing data containing up to 35 kilobases long reads. Different minimum match lengths were selected. The results showed that our GPU acceleration is up to 9x faster for computing MEM compared to the fastest CPU implementation running on a server-grade machine with 24 threads. Computing suffix array intervals is up to 3x faster, whereas calculating the location of the match is up to 9x faster. The implementation is publicly available at: https://github.com/nahmedraja/GPUseed

7 REFERENCES

[1] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[2] S. F. Altschul et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[3] A. L. Delcher et al. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[4] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *In IEEE FOCS 2000*, FOCS '00, pages 390–398, 2000.

[5] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv*, May 2013.

[6] Y. Liu and B. Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, 2012.

[7] G. Miclotte et al. Jabba: hybrid error correction for long sequencing reads. *Algorithms for Molecular Biology*, 11(1):10, May 2016.

[8] F. Fernandes and A. T. Freitas. slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array. *Bioinformatics*, 30(4):464–471, 2014.

[9] M. Vyverman et al. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

[10] Y. Liu and B. Schmidt. CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing. *Design Test, IEEE*, 31(1):31–39, Feb 2014.

[11] M. C. Schatz and C. Trapnell. Fast Exact String Matching on the GPU. Technical report, University of Maryland, 2007.

[12] A. Abu-Doleh, K. Kaya, M. Abouelhoda, and Ü. V. Çatalyürek. Extracting Maximal Exact Matches on GPU. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1417–1426, May 2014.

[13] J. Pantaleoni and N. Subtil. NVBIO. nvlabs.github.io/nvbio/, 2015.

[14] T. W. Lam et al. High Throughput Short Read Alignment via Bi-directional BWT. In *IEEE BIBM 2009*, pages 31–36, Nov 2009.

[15] H. Li. Burrows Wheeler Aligner version 0.5.9. github.com/lh3/bwa/releases/tag/bwa-0.5.9. Accessed 1 January, 2019.

[16] N. Ahmed et al. GPU Accelerated API for Alignment of Genomics Sequencing Data. In *IEEE BIBM 2017*, pages 510–515, Nov 2017.

[17] H. Li. Exploring single-sample SNP and indel calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, Jul 2012.

[18] J. Zhang et al. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *IEEE/ACM CCGrid 2013*, May 2013.

[19] D. Merrill. CUB - a configurable C++ template library of high-performance CUDA primitives. nvlabs.github.io/cub. Accessed 2 January, 2019.

[20] Pacbio. datasets.pacb.com/2013/Human10x/READS/2530572/0001/Analysis_Results/m130929_024849_42213_c100518541910000001823079209281311_s1_p0.1.subreads.fasta, 2014.