Extracting Maximal Exact Matches on GPU

Anas Abu-Doleh[†‡], Kamer Kaya[†]
[†]*Dept. of Biomedical Informatics*
[‡]*Dept. of Electrical and Computer Eng.*
The Ohio State University
abudoleh.1@osu.edu,
kamer@bmi.osu.edu

Mohamed Abouelhoda
Faculty of Engineering
Cairo University, Giza, Egypt
mabouelhoda@yahoo.com

Ümit V. Çatalyürek
Dept. of Biomedical Informatics
Dept. of Electrical and Computer Eng.
The Ohio State University
umit@bmi.osu.edu

***Abstract*—The revolution in high-throughput sequencing technologies accelerated the discovery and extraction of various genomic sequences. However, the massive size of the generated datasets raise several computational problems. For example, aligning the sequences or finding the similar regions in them, which is one of the crucial steps in many bioinformatics pipelines, is a time consuming task. Maximal exact matches have been considered important to detect and evaluate the similarity. Most of the existing tools that are designed and developed to find the maximal matches are based on advanced index structures such as suffix tree or array. Although these structures triggered the development of efficient search algorithms, they need large indexing tables which yield large memory footprint for the software using them and bring significant overhead. In this article, we introduce a novel tool GPUMEM which effectively utilizes the massively parallel GPU threads while finding maximal exact matches inside two genome sequences using a lightweight indexing structure. The index construction, which is also handled in GPU, is so fast that even by including the index generation time, GPUMEM can be faster in practice than a state-of-the-art tool that uses a pre-built index.**

***Keywords*-maximal exact matches; GPUs; parallel programming; indexing;**

I. INTRODUCTION

In the last decade, there have been exciting advancements in high-throughput sequencing technologies which massively improved the size of the sequencing data and reduced the costs. However, these also yield new computational problems for the researchers who are interested in processing this data. Today, evaluating and/or quantifying the similarity of the sequences and aligning them with each other is considered as a highly computation-intensive task in many bioinformatics pipelines [7]. Hence, there has been a growing interest on solving this problem.

Finding the optimum similarity, e.g., the best local alignment via Smith-Waterman [15], may not be possible for large sequence datasets. For this reason, heuristic approaches have been proposed along-with novel techniques to improve their speed and accuracy. In general, these heuristic approaches extract the shared regions from the sequences and use them as anchors for the next step of a full alignment process. For example, BLAST [3] extracts fixed-size seeds that are common in the sequences. However, using a fixed seed length can be problematic because a massive amount of seeds can be generated from a single long sub-sequence that exists in all the sequences. This problem has attracted more attention, and another approach, using maximally matched regions in the sequences have been investigated.

The problem of extracting a *maximal unique match* (MUM) was introduced in [6] and a tool MUMmer has been made available to public. A MUM is a sub-sequence which appears only once in both sequences (i.e., unique) and cannot be extended to either directions without having a mismatch (i.e., maximal). Other forms of the problem such as extracting maximal *rare* matches have also been studied [14]. The problem we will study on in this paper is the *maximal exact match* (MEM) extraction when a *reference* and a *query* sequence are given. In this variant, the uniqueness (or rareness) is not a constraint. The problem is interesting especially when the number of MUMs is low, which is usually the case in practice, extracting all the MEMs is sufficient to produce anchors between two sequences. Furthermore, finding MEMs is an essential task while solving various bioinformatics problems such as mapping long reads [13], genomic assembly [10], and whole genome comparison [5].

In this paper, we introduce a very fast GPU-based tool GPUMEM whose good matching performance is based on the efficient utilization of the massively parallel threads. Instead of complex and large index structures employed by the existing tools, GPUMEM uses a lightweight GPU-based parallel indexing technique which is more suitable for memory restricted devices. Experiments on real-life genomic sequences show that the GPU-based parallel indexing is so fast that the combined indexing and matching method is faster than a state-of-the-art tool that uses an already pre-built index. To alleviate the memory restriction problem, the tool partitions the 2D (reference/query) search space into equally sized regions and first solves these subproblems by employing a novel load balancing heuristic. To the best of our knowledge, GPUMEM is the first GPU-based

Table I
NOTATION USED IN THE PAPER

$\mathcal{R}$	Reference sequence
$\mathcal{Q}$	Query sequence
(r, q, λ)	A maximal exact match of length λ in $\mathcal{R}$ and $\mathcal{Q}$ $\mathcal{R}_r \cdots \mathcal{R}_{r+\lambda-1} = \mathcal{Q}_r \cdots \mathcal{Q}_{q+\lambda-1}$
L	Minimum length of an exact match
ℓ_s	Indexing seed length
Δ_s	Indexing step size
n_r	#row tiles in 2D work partitioning
n_c	#column tiles in 2D work partitioning
ℓ_{tile}	size of a (square) tile
n_{block}	#blocks in a tile
ℓ_{block}	width of block
τ	#threads
w	#query locations per thread

tool designed and developed for the maximal exact match extraction problem.

The rest of the paper is organized as follows: Section II introduces the notation, formally defines the MEM extraction problem, and provides necessary information on the existing tools and techniques. The proposed method is described in Section III. Section IV evaluates the performance of GPUMEM by comparing it with existing tools in the literature. Section V concludes the paper and discusses possibilities for future work.

II. NOTATION AND BACKGROUND

Let $\mathcal{R}$ be a (*reference*) sequence of the letters in the alphabet Σ. Let $\mathcal{Q}$ be another (*query*) sequence also on Σ and L be an integer. The maximal exact match extraction problem is defined as follows: given $\mathcal{R}$, $\mathcal{Q}$, and L, we want to find all triplets (r, q, λ) such that

- $\mathcal{R}_{r+i} = \mathcal{Q}_{q+i}$ for $i = \{0, \cdots, \lambda - 1\}$ and $\lambda \geq L$,
- $\mathcal{R}_{r-1} \neq \mathcal{Q}_{q-1}$ and $\mathcal{R}_{r+\lambda} \neq \mathcal{Q}_{q+\lambda}$,

where $\mathcal{R}_i \in \Sigma$ ($\mathcal{Q}_i \in \Sigma$) denotes the i-th letter of $\mathcal{R}$ ($\mathcal{Q}$). In this work, we are interested in genomic sequences. Hence, for the rest of the paper we assume the *base pairs* are coming from $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$. The notation we use in the paper is summarized in Table I.

A. Related Work

Searching and aligning sequences or finding their common sub-sequences have been extensively studied in the literature and efficient data structures and algorithms have been proposed. The well known *suffix tree* has been considered as one of these structures and linear algorithms to search the matches between sequences by using a suffix tree already exist [4]. In general, all the existing MEM extraction algorithms index the reference sequence and use this index to efficiently find MEMs between the reference and a query sequence. One of the first examples of efficiently using the suffix tree to find maximal matches is presented in [6]. However, the major drawback of this approach is the large indexing tables. Reformulating the suffix tree in order to reduce the memory usage is also introduced in the literature and an *enhanced suffix array* data structure has been proposed [2]. Another data structure, *sparse suffix array* has been proposed for the same purpose [11].

In addition to the data structures, several publicly available tools that improve the MEM extraction speed now exist for practical purposes. MUMmer [6], which is a well-known suffix-array-based alignment tool, has been enhanced to find MEMs [12]. Another tool sparseMEM uses the sparse suffix array to reduce its memory footprint [11]. But this reduction also introduces more computational work. To improve the performance of sparseMEM, a similar tool essaMEM uses auxiliary sparse data structures [16]. Furthermore, the parallel (shared memory) performance of essaMEM is significantly better than sparseMEM. Another maximal exact match extraction tool slaMEM uses the backward search method employed in the well-known *FM-Index* [9] while searching longest common prefixes [8].

B. GPU architecture

A graphical processing unit (GPU) is a highly parallel device built to speed up the execution of the massively computational applications. The single instruction-multiple threads (SIMT) architecture of the GPU is based on that all threads in the same core execute the same instruction in parallel. In a GPU, a *thread* defines the finest computational granularity throughout the execution. Each thread has a relatively large number of registers and some thread local (off-chip) memory in case registers are not enough. The threads are grouped within *blocks*, and all the threads in a single block can access the same shared memory. The maximum number of threads a block can contain is limited. A *grid* of blocks is responsible form whole kernel execution. Thus, during the computation, each thread has a unique ID in a block, and each block has a unique ID within a grid.

A GPU is composed of streaming multiprocessors (SM) and each block is assigned to an SM (having 192 CUDA cores) within the SM's execution capacity. A group of threads that physically run in parallel is called a warp. The number of such threads, i.e., the warp size, is currently 32 for cutting-edge GPU architectures. Each thread in a warp usually operates on a different data but they execute the same instruction in parallel. Hence, to obtain the best performance, the distribution of the work assigned to the threads in a warp need to be balanced. In addition, the computations within a warp must be as homogeneous as possible because, the threads in the same warp are serialized if they are executing different instructions. When the kernel has '**if**'s and '**else**'s, i.e., when the computation is branched (also called *divergent* in GPU computing), the threads are serialized, concurrency decreases and performance degrades.

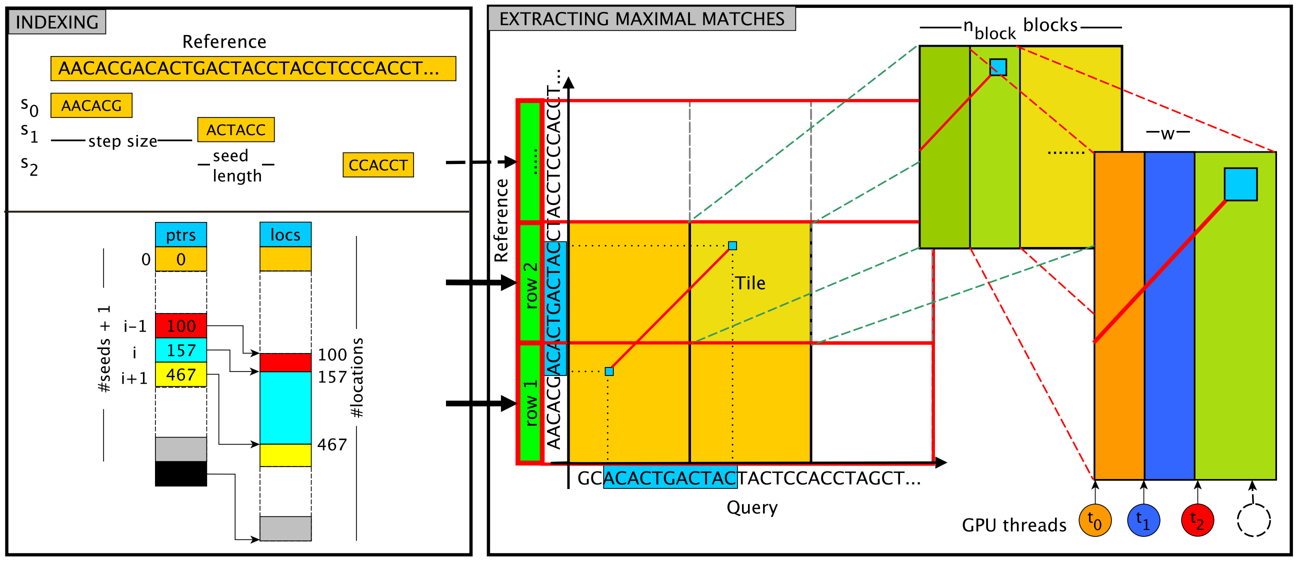


Figure 1. A high-level structure of GPUMEM's MEM extraction algorithm (right) and its simple index structure containing two basic arrays (left).

III. Finding maximal exact matches in GPU

GPUMEM effectively utilizes the massively parallel GPU threads while indexing the reference and extracting maximal exact matches in two sequences without employing complex indexing tables. A high-level structure of GPUMEM is shown in Figure 1; given $\mathcal{R}$ and $\mathcal{Q}$, it partitions the 2D reference/query space, which is the memory layout with $\mathcal{R}$ as the y-axis and $\mathcal{Q}$ as the x-axis, into multiple $\ell_{tile} \times \ell_{tile}$ square tiles, which correspond to smaller sub-problems, to fit the problem to GPU memory. While searching for MEMs, starting from the lowest tile-row, GPUMEM processes all the tiles in a row with an index constructed only from the corresponding reference region. That is only a partial index is created for ℓ_{tile} base pairs of reference, loaded into the memory, and used until all the tiles in that row are processed.

As the figure shows, GPUMEM also divides each tile into n_{block} rectangular blocks with dimensions $\ell_{tile} \times \ell_{block}$. Hence $\ell_{tile} = n_{block} \times \ell_{block}$. Each of these blocks will be assigned to a single GPU block where each thread is originally responsible for w consecutive query locations/seeds. Hence, $\ell_{block} = \tau \times w$ and the query seeds at location $i < w$ will be concurrently processed with the query seeds at locations $i + kw$ for $1 \leq k \leq \lfloor \ell_{tile}/w \rfloor$. The variable i incremented by one when each set of τ query locations are processed. Here, we describe the steps taken during execution in detail.

A. Index construction

To generate its lightweight index, GPUMEM uses seeds of length ℓ_s and simply stores the seed locations in the reference. As the bottom-left part of Figure 1 shows, the index contains two arrays: an array locs which stores the locations of indexed seeds in *sorted order*, and another one ptrs which stores the prefix-sum of the number of occurrences for each seed, i.e., the start location for each seed in the locs array. The last element of ptrs is |locs|. Hence, all the locations of a seed s are the locations in between locs[ptrs[s]] and locs[ptrs[$s+1$] - 1].

With a simple approach using a full index, a seed can start anywhere in $\mathcal{R}$, hence there are $|\mathcal{R}|$ locations that need to be stored. Assuming each location value is 32 bits, with a 1Gbp (base pairs) reference, we need 4GB memory only for the locs array to store all the locations. Obviously, this is a burden for a restricted memory device. In addition to its memory footprint, the index construction time can also be a problem especially for one-time-use long reference sequences. To alleviate these problems, similar to the spareness factor used in sparseMEM and essaMEM, GPUMEM uses a sparsification parameter, *step size*, denoted by Δ_s to reduce the index size and construction time. The step size is a distance between the indexed seeds of $\mathcal{R}$. Note that one cannot increase the step size unlimitedly: to guarantee the existence of at least one matched seed location for all the maximal exact matches of length at least L, we need

$$\Delta_s \leq L - \ell_s + 1, \quad (1)$$

where L is the parameter that sets the threshold on the length of desired MEMs. We use the maximum possible value for Δ_s. With this optimization, the number of locations one needs to store reduces to $|\mathcal{R}|/\Delta_s$ from $|\mathcal{R}|$. Hence, when $\Delta_s = 1$, GPUMEM uses a full index. Note that using distanced seeds will also incur a computational overhead since GPUMEM needs to expand each matched seed location for maximality. However, it also reduces the number of matched points where many will overlap and will be unneccesary. As the experiments will show, the proposed approach will perform well thanks to the massive parallelism of this overhead provided by the device.

As Figure 1 shows, GPUMEM does not construct and store all its index structure at once in the memory. Instead, it partitons the 2D-search space into multiple tiles, and at a time, it only uses the index of reference region corresponding to the current row. Hence, only $n_{locs} = \lceil \ell_{tile}/\Delta_s \rceil$ locations are stored in the locs array at once where ℓ_{tile} is the width and height of a square tile. Hence, the locs array can be stored in $n_{locs} \times \lceil \log_2 \ell_{tile} \rceil$ bits. To represent the seeds, each base is represented by 2 bits (i.e., A $= 00$, C $= 01$, G $= 10$, and T $= 11$). Hence, each seed is represented by $2 \times \ell_s$ bits and the number of entries in the ptrs array is $2^{2\ell_s} = 4^{\ell_s}$. So in theory, the total memory required for ptrs is $4^{\ell_s} \lceil \log_2 n_{locs} \rceil$.

Algorithm 1 shows the pseudocode for GPU-based index construction which is done in four steps: the first step is counting the occurrences of the indexed seeds and filling the ptrs array. To avoid conflicts, GPUMEM uses the `atomicAdd`(mem, val) operation which atomically increases the content of mem by val and returns the old value. In the second step, a prefix-sum operation is performed over ptrs to find the start point of each seed in the locs array which is filled in the next step by using a temporary array temp. This time, atomic increments on temp are used to reserve a place on locs. Since, the array is filled in parallel, probably, the seed locations will not be fully sorted. In the last step GPUMEM sorts them by assigning a thread for each seed.

Algorithm 1: Partial index construction

```
input  : $\mathcal{R}$, $start$, $end$
output: ptrs [.], locs [.]

Step 1: Compute seed counts
for each seed $s$ in parallel do
    ptrs[$s$] $\leftarrow 0$

for each location $l \in [start, end]$ of $\mathcal{R}$ in parallel do
    $s \leftarrow (\mathcal{R}_l, \mathcal{R}_{l+1}, \cdots, \mathcal{R}_{l+\ell_s-1})$
    atomicAdd(ptrs[$s+1$], 1)

Step 2: Prefix-sum over the ptrs array
GPUPrefixSum(ptrs)

Step 3: Fill the locs array
for each seed $s$ in parallel do
    temp[$s$] $\leftarrow$ ptrs[$s$]

for each location $l \in [start, end]$ of $\mathcal{R}$ in parallel do
    $s \leftarrow (\mathcal{R}_l, \mathcal{R}_{l+1} \cdots, \mathcal{R}_{l+\ell_s-1})$
    $index \leftarrow$ atomicAdd(temp[$s$], 1)
    locs [$index$] $\leftarrow l$

Step 4: Sort the locations in locs
for each seed $s$ in parallel do
    sort(locs [ptrs[$s$], $\cdots$, ptrs[$s+1$] $- 1$])
```

B. Extracting MEMs inside the blocks

As described above, each $\ell_{tile} \times \ell_{block}$ block is assigned to a GPU block during the execution. The blocks are processed in four steps: first, a proactive heuristic is performed to balance the work of GPU threads during the MEM extraction process. Second, each thread generates and expands the exact matches associated with the query location (seed) they are responsible for. Then, a parallel *combine* algorithm is applied to merge consecutive matches. Finally, a *filter* operation is performed in order to generate two types of exact matches: *in-block MEMs* which do not touch the block boundaries, *out-block MEMs* which touch them. Note that an in-block MEM is a real maximal exact match however, an out-block MEM is only maximal within the block boundaries and can be expanded further when the boundaries are passed.

1) Proactive load-balancing heuristic: There are three main tasks performed by each GPU thread: generating, expanding, and combining exact matches. The workload due to expansion is usually well-balanced among the GPU threads. However, the amount of the work for generating and combining the exact matches varies since it depends on the occurrence of the query seed associated with each thread location. Since these sequences are not random, i.e., the bases in the sequences are not chosen from a uniform random distribution, the number of appearances for two query seeds inside the indexed reference region can be significantly different. Furthermore, without load balancing, some threads can stay idle because the corresponding query seed does not exist in the index.

To make the extraction process more efficient, GPUMEM applies a proactive load-balancing technique which fairly assigns all the idle threads to the query locations with a large workload. Hence, a group of threads will be assigned to a single query location and all the work due to that location will be divided among them as evenly as possible. Figure 2 shows an example of how GPUMEM reassigns the threads according to the work distribution among the query seeds.

The pseudocode of the load-balancing heuristic is given in Algorithm 2. GPUMEM uses two temporary arrays of size τ; task and load. Consider the threads are numbered from 1 to τ; after the prefix-sum operations in the algorithm, the value task[i] is the number of threads whose thread ids are at most i and whose corresponding seeds (due to original thread/seed assignment) appear in the index at least once. The value load[i] is the total load, i.e., total number of match locations, for these threads. By using these auxiliary arrays, the load-balancing heuristic fills two arrays assign and group of size $\tau + 1$ and τ, respectively, in parallel.

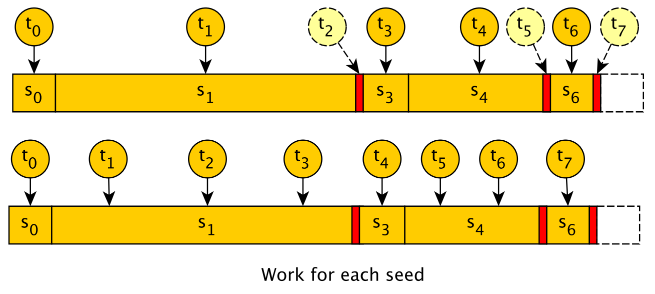


Figure 2. A toy example for proactive load balancing: the original straightforward thread/query-seed assignment is given above. Below, the distribution after the load-balancing heuristic is given. Note that a single thread will be responsible for only a single query location.

Let $T_{idle} = \tau - \mathsf{task}[\tau]$ be the number of threads that will remain idle due the original assignment. Let $T_{load} = \mathsf{load}[\tau]$ be the total load. Algorithm 2 cleverly and concurrently assigns the idle threads to the seeds in parallel by using the cumulative distribution of the overall work upto the current task/seed, i.e., $\mathsf{load}[.]/T_{load}$. It fills an array assign such that $\mathsf{assign}[k+1] - \mathsf{assign}[k]$ for the kth seed in the current block. That is if $\mathsf{assign}[k] = 5$ and $\mathsf{assign}[k+1] = 7$ thread 5 and thread 6 will work for the corresponding seed which was originally assigned to thread $k+1$. After the assignment is done, each thread tid finds its assigned seed, i.e., $\mathsf{group}[tid]$ by using a binary search on the assign in parallel.

Algorithm 2: Proactive load-balancing heuristic

```
output: assign: prefix-sum array of balanced workload
for each thread $tid$ in parallel do
    $s \leftarrow$ the query seed originally assigned to $tid$
    load $[tid] \leftarrow$ ptrs$[s+1]$ − ptrs$[s]$
    if load$[tid] > 0$ then
        task$[tid] \leftarrow 1$
    else
        task$[tid] \leftarrow 0$
GPUPrefixSum (load)
GPUPrefixSum (task)
$T_{load} \leftarrow$ load$[\tau]$
$T_{idle} \leftarrow \tau -$ task$[\tau]$
assign$[0] \leftarrow 1$
for each thread $tid$ in parallel do
    $offset \leftarrow T_{idle} \times$ (load$[tid]/T_{load}$)
    assign$[tid+1] \leftarrow$ task$[tid] + offset$
for each thread $tid$ in parallel do
    group$[tid] \leftarrow$ binarySearch(assign, $tid$)
```

2) Generating exact match triplets: After the load balancing heuristic, each thread group has an assigned query seed/location q which is in a distance of a multiple of w from the next thread group's location. The threads generate the initial match triplets (r, q, λ) where r is the reference location, and λ is the length of the match. Initially, λ is set to ℓ_s, i.e., the size of a seed, for all the exact match triplets.

To make the connections between the triplets of two consecutive thread groups and combine them later, if $\ell_s < w$, GPUMEM extends the match triplets to right. That is for a triplet (r, q, ℓ_s), a pairwise seed-by-seed comparison is performed starting with the reference and query seeds generated from the locations $\mathcal{R}_{r+\lambda}$ and $\mathcal{Q}_{q+\lambda}$, respectively. If the seeds match an additional ℓ_s is added to λ and the process continues. The extension stops when a mismatch is found or the triplet length reaches w. To extract all the valid MEMs and not to extract a MEM more than once, GPUMEM uses $w = \Delta_s$.

3) Combining exact match triplets: After generating the triplets, GPUMEM combines them by checking the pair-wise overlaps between the triplets of two thread groups. Two triplets (r, q, λ) and (r', q', λ') overlap if

$$0 < (r' - r) = (q' - q) \leq \lambda.$$

If this is the case, the first triplet (r, q, λ) is replaced by $(r, q, r' - r + \lambda')$ and the second triplet is deleted (in practice GPUMEM just sets λ' to zero). The combining process is performed in parallel as shown in Algorithm 3.

As explained above, after the load balancing, each GPU thread is a member of a group corresponding to a query seed that appears at least once in the index. There can be many exact match triplets for that seed. GPUMEM uniformly assigns these triplets to the threads in the group and each thread is responsible from its own triplets during the combine process. The overlapping triplets are combined as explained above in a pair-wise manner.

To avoid possible conflicts due to parallelization, the process works in $2k - 1$ iterations where $k = \log_2 \tau$. Let $s_0, s_1, \cdots, s_{\tau-1}$ be the seeds being processed in the current block and let $|i - j|$ be the *distance* between the seeds s_i and s_j. At each iteration, only a subset of the thread groups (or seeds) are active, and the triplets of an active seed s_i will be combined only with the triplets of the seed s_{i+d}. The combine process starts with $d = 1$ at iteration 1, and till the kth iteration, d is doubled. After that, d is halved when an iteration is completed. Hence, for the last one, i.e., $2k - 1$th iteration, $d = 1$. The selection of the active seeds also follow a similar pattern: for the first k iterations, the seeds s_is where $i \bmod 2d = 0$ are active. For the rest of the process, an s_i is active if and only if $i > d$ and $i \bmod 2d = d$. It is not hard to verify that all the overlapping match triplets are found and reduced to a single triplet that starts from the left most location when the process is completed. Figure 3 shows an example of the combining pattern for 16 seeds/threads.

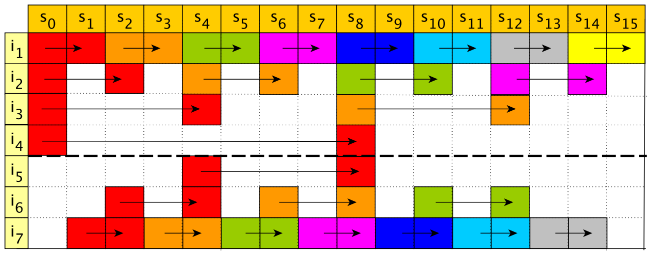
Algorithm 3: Combining exact match triplets

```
$d \leftarrow 1$
$k \leftarrow \log_2 \tau$
for $iter = 1$ to $2k - 1$ do
    for each thread $tid$ in parallel do
        $src \leftarrow \mathsf{group}[tid]$
        $ctrl \leftarrow src$
        if $iter \geq k$ then
            $ctrl \leftarrow ctrl - d$
        if $ctrl \geq 0$ and $ctrl \bmod 2d = 0$ then
            $trgt \leftarrow src + d$
            if $trgt < \tau$ then
                $\mathcal{S}$: the triplets of the seed $s_{src}$
                $\mathcal{S}_{tid}$: the triplets in $\mathcal{S}$ assigned to $tid$
                $\mathcal{T}$: the triplets of the seed $s_{trgt}$
                for each $(r, q, \lambda) \in \mathcal{S}_{tid}$ do
                    for each $(r', q', \lambda') \in \mathcal{T}$ do
                        $\delta_r = r' - r$
                        if $\delta_r = q' - q$ and $\delta_r \leq \lambda$ then
                            $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(r, q, \lambda)\}$
                            $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(r', q', \lambda')\}$
                            $\mathcal{S} \leftarrow \mathcal{S} \cup \{(r, q, \delta_r + \lambda')\}$
    if $iter < k$ then
        $d \leftarrow d \times 2$
    else
        $d \leftarrow d/2$
```

In the proposed approach, the distance between two active seeds is always $2d$ at each iteration. Considering the combine distance is d, the selection of active seeds guarantees that each overlapping triplet will be either modified or deleted but these cases cannot be at the same iteration. Hence, there cannot be a read/write conflicts during the parallel execution.

After finishing the combine process, GPUMEM expands the remaining triplets with a positive length to the left using a seed-by-seed comparison within the reference and the query. The expansion process is similar to the one explained above which does the same to the right.

4) Finding in-block and out-block MEMs: After the matches are combined for each execution, the sets of in-block and out-block MEMs are constructed. In this step, each GPU thread processes a triplet and expands it to the right and the left seed by seed until a mismatch is found or the block boundaries are reached. Then the triplets of length at least L are filtered and the in-block and out-block MEMs are constructed. The first set is transferred to the host for reporting and the second is kept on the device for further processing.



s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15}
i_1
i_2
i_3
i_4
i_5
i_6
i_7

Figure 3. A toy combine example with 16 seeds/threads and hence 7 iterations: each combine operation is shown with an arrow where the arrow goes from the active seed to the target seed whose triplets will be merged with the active threads triplets. Each such seed pair is colored with different color at each iteration.

C. Finding in-tile and out-tile MEMs

Similar to the previous step, the *in-tile* and *out-tile* MEMs are generated in this phase. The input of the process is the union of the *out-block* MEMs constructed obtained from the blocks of the current tile.

1) Combining out-block MEMs: In this step, two overlapping match triplets are combined as also described in Section III-B3. Although the mathematical equations used to detect an overlap and perform the combine operation are the same, the algorithm is different. In this part, for each $\ell_{tile} \times \ell_{block}$ block, GPUMEM first performs a parallel sort on their out-block MEMs; each triplet (r, q, λ) is sorted in the increasing order of the $r - q$ values. In case of equality, the value q is used to sort the triplets. This operation places the overlapping triplets in each $\ell_{tile} \times \ell_{block}$ block consecutively in the sorted order. Each thread is assigned to $\ell_{tile} \times \ell_{block}$ block and performs the combine operation. After that the in-tile and out-tile MEMs are constructed. The in-tile MEMs are moved to the host for reporting. The out-tile triplets are inserted to a global list which contains all the out-tile triplets found during scanning the whole reference and query sequence.

2) Combining out-tile triplets: We observed that the number of out-tile triplets is much less considered to out-block ones. Considering the size of a tile is $1K \times \tau \times \Delta_s$, this is expected. The short list of out-tile triplets is transferred to the host CPU and a sequential merge-sort operation is performed to sort the list with respect to the $r - q$ values as we did before. GPUMEM performs a simple scan over this list to obtain the final (and the longest) MEMs.

IV. EXPERIMENTAL RESULTS

For GPUMEM experiments, we used a machine equipped with an Intel core i7-960 CPU clocked at 3.2Ghz having 4 cores and 24GB of memory. The machine has an NVIDIA Tesla K20c GPU featuring 13 Streaming Multiprocessors (SM), 192 cores per SM clocked at 700 MHz (for a total of 2496 CUDA cores), and 4.8GB of global memory clocked at 2.6 GHz. ECC is enabled. On the software side, the machine runs CentOS 5.8 with Linux 2.6.18. The code was compiled using CUDA 5.0 and gcc 4.2.4.

To run the CPU-based tools, we used a machine with dual Intel Xeon E5520 Quad-core CPUs (with 2-way Simultaneous Multithreading, and 8MB of L3 cache per processor) and 48GB of main memory. To parallelize sparseMEM and essaMEM, we used up to 8 threads (single thread/core, i.e., no HyperThreading). The tools were run on CentOS 6, and compiled with GCC 4.5.2 using the -O2 optimization flag.

We used several genomic sequences to evaluate GpuMem's performance and compare it with that of the existing, publicly available tools. These sequences have also been used in the literature in studies which focus on the MEM extraction problem, e.g., [1]. The sequences, their lengths, and explanations are given in Table II. To represent and store the sequences in memory, we apply a common technique as described in the previous section and encode the sequences using 2 bit per base.

Table II
Data files used in the experiments. Sequence lengths are given in million base pairs (Mbp).

Name	Length	Description
chr2h	242.97Mbp	Human chromosome 2
chrI	233.10Mbp	Saccharomyces cerevisiae chrI
chr1m	195.75Mbp	Mouse chromosome 1
chrXh	154.12Mbp	Human chromosome X
chrXc	133.55Mbp	Chimpanzee chromosome X
dmelanogaster	23.30Mbp	Drosophila melanogaster chr. 2L
EcoliK12	4.71Mbp	Escherichia coli chromosome K12
chrXII	1.09Mbp	Saccharomyces cerevisiae chrXII

A. *Performance analysis of* GpuMem

We first investigate how GpuMem's MEM extraction performance changes with respect to the problem parameters, i.e., query size and L, respectively. As Fig. 4 shows, the extraction time increases linearly with the query size. This is somehow expected since the number of possible match start location pairs, i.e., the problem size, is $|\mathcal{R}| \times |\mathcal{Q}|$. Yet, it is still questionable if the increase on the MEM extraction time is due to the increase on the query size or the number of MEMs extracted by GpuMem that also increases with $|\mathcal{Q}|$. Indeed, when $|\mathcal{Q}|$ increases, as the figure shows (on the right y-axis), the number of actual MEMs also increases proportional to the execution time.

To further investigate its performance, we run GpuMem on the same reference/query pair but with different L values. Figure 5 shows the results of this experiment. Since $|\mathcal{R}|$ and $|\mathcal{Q}|$ stay constant, as L increases, the decrease on the MEM extraction time can be explained by the decrease on number MEMs extracted. However, as the figure shows, these two values do not decrease with the same pace: although the decrease on the execution time for $L = 20$ and $L = 30$ is faster compared to the decrease on the number of extracted MEMs, after $L = 50$, the latter is faster than the former.

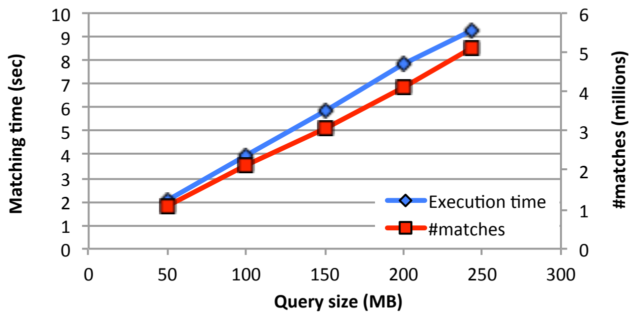


Figure 4. MEM extraction of GpuMem with respect to the query size: chr1m is used as the reference and the first 50Mbp, 100Mbp, 150Mbp, 200Mbp of chr2h as well as the whole 242.97Mbp sequence are used to generate the data points. In the figure, the first y-axis (left) shows the execution time (blue marker). The second one (right) shows the number of extracted MEMs by GpuMem with $L = 50$ (red line).

Hence, we can conclude that the MEM extraction time is affected by both the input size (reference/query length), and the output size (the number of MEMs extracted).

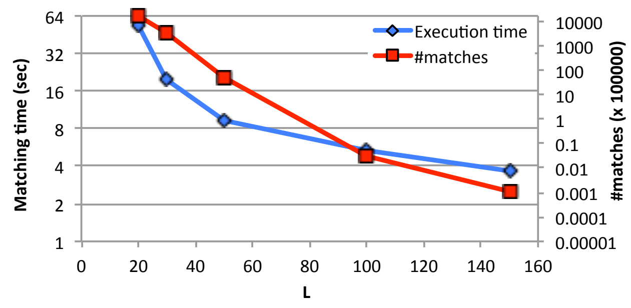


Figure 5. MEM extraction performance of GpuMem with respect to L: chr1m and chr2h are used as the reference and query sequences, respectively, with $L \in \{20, 40, 50, 100, 150\}$. In the figure, the first y-axis (left) shows the execution time (blue line). The second axis (right) shows the number of extracted MEMs (red line). Both axis are in log-scale.

B. *Comparing* GpuMem *with the existing tools*

We compared the performance of GpuMem with that of four state-of-the-art tools; sparseMEM [11], essaMEM [16], MUMmer [6], [12], and slaMEM [8]. Among these tools, sparseMEM and essaMEM support shared-memory parallelism and we run them for $\tau = 1$, 4, and 8 threads. The I/O times for all these tools are not included in the tables. Although the input and output, i.e., reference/query and the MEMs, should be the same, the implementions to read the input and write the output, hence the I/O times, can vary. Obviously, they are not related with the MEM extraction performance.

Table III shows the index generation times of the tools in seconds for nine different reference, query, and L configurations. The first three columns of the table describe the configuration. As the table shows, for the reference chr1m

Table III
INDEX GENERATION TIMES (SECS) OF THE TOOLS USED IN THE EXPERIMENTS FOR VARIOUS REFERENCE/QUERY PAIRS AND L VALUES. THE SEED LENGTH $\ell_s = 13$ FOR GPUMEM EXCEPT THE LAST ROW WHERE $\ell_s = 10$ IS USED. I/O TIMES ARE NOT INCLUDED.

			sparseMEM			essaMEM			MUMmer	slaMEM	GPUMEM
Reference	Query	L	$\tau = 1$	$\tau = 4$	$\tau = 8$	$\tau = 1$	$\tau = 4$	$\tau = 8$			
		100									1.41
chr1m	chr2h	50	73.84	37.17	28.51	75.08	41.67	30.68	99.58	278.32	2.51
		30									5.58
chrXc	chrXh	50	48.78	24.84	18.37	49.72	27.70	19.87	66.42	169.95	1.74
		30									3.11
dmelanogaster	EcoliK12	20	7.74	3.66	2.38	8.34	4.27	2.69	10.73	39.71	1.20
		15									3.19
chrXII	chrI	20	0.22	0.09	0.10	0.31	0.13	0.13	0.26	1.68	0.38
		10									0.05

Table IV
EXECUTION TIMES (SECS) OF THE MATCHER TOOLS USED IN THE EXPERIMENTS TO FIND MAXIMAL EXACT MATCHES FOR VARIOUS REFERENCE/QUERY PAIRS AND L VALUES. THE SEED LENGTH $\ell_s = 13$ FOR GPUMEM EXCEPT THE LAST ROW WHERE $\ell_s = 10$ IS USED. I/O TIMES ARE NOT INCLUDED.

			sparseMEM			essaMEM			MUMmer	slaMEM	GPUMEM
Reference	Query	L	$\tau = 1$	$\tau = 4$	$\tau = 8$	$\tau = 1$	$\tau = 4$	$\tau = 8$			
		100	163.75	444.72	502.00	161.91	14.49	10.14	159.17	84.56	5.38
chr1m	chr2h	50	164.42	443.24	499.13	161.00	59.29	34.89	161.86	84.86	9.24
		30	213.32	460.08	507.95	211.70	116.12	32.00	312.28	100.16	20.19
chrXc	chrXh	50	70.19	187.22	223.38	68.78	42.99	24.91	78.65	52.36	5.86
		30	111.79	197.61	232.65	110.13	83.13	25.58	163.58	80.77	11.22
dmelanogaster	EcoliK12	20	3.22	7.32	4.76	3.21	0.36	0.32	2.68	1.54	0.08
		15	3.25	7.57	6.46	3.24	0.71	2.68	2.75	1.57	0.24
chrXII	chrI	20	0.08	0.13	0.08	0.08	0.01	0.01	0.08	0.06	0.01
		10	0.13	0.25	2.34	0.13	0.08	2.19	0.14	0.11	0.02

and query chr2h, GPUMEM's index generation is 20.2, 11.4, and 5.2 times faster when $L = 100$, 50, and 30, respectively, compared to the essaMEM with 8 threads, which is the best CPU-based tool for overall execution time in almost all the experiments. As the table shows, the index generation of GPUMEM becomes slower as L decreases since the step size Δ_s, hence the number of necessary index locations to extract all MEMs increases. The only exception to this is the last row of the table that happens due to reduction of ℓ_s from 13 to 10 since it is necessary to use a smaller or equal length seeds when $L = 10$. Unlike GPUMEM, the index generation time for the existing CPU-based tools is not affected by L. However, for all the configurations in the table, GPUMEM's index construction takes less than 5.6 seconds whereas the state-of-the-art tools can take up to hundreds of seconds (especially if they have no support for parallelism).

The MEM extraction performance of GPUMEM is compared with the existing CPU-based tools in Table IV for the same nine reference/query/L configurations. In short, GPUMEM is significantly faster than all the tools in our experimental setting. As expected, when L decreases, the MEM extraction time usually increases not only for the GPU-based tool but also the CPU-based tools. Although the impact of L on the extraction time seems to be more for GPUMEM, even with this, for the reference/query sequences chrXc/chrXh, the proposed tool is 4.3 and 2.3 times faster than the best existing tool (essaMEM with 8 threads) with $L = 50$ and 30, respectively. Furthermore, thanks to the simplicity of indexing and its suitability to the GPU, the difference between the MEM extraction time of GPUMEM and essaMEM is larger than the index generation time of GPUMEM for (relatively) longer reference/query sequences (the first 5 configurations). Hence, for these cases, GPUMEM even both with indexing and extraction is faster than an existing tool using a prebuilt index.

Although essaMEM and sparseMEM are based on similar algorithms, the main focus of sparseMEM is decreasing the memory footprint of the MUMmer. sparseMEM exploits the multi-core CPUs to reduce the index size which yields an increase on the MEM extraction time as also shown in [11]. Hence, the input to sparseMEM with different τ values are not the same, i.e., when τ increases, the index gets smaller and the problem becomes harder. Note that GPUMEM already uses a lightweight index with a memory footprint low enough to fit in a memory-restricted device such as a GPU.

C. Impact of load balancing

We investigated the impact of load balancing on the MEM extraction performance of GPUMEM. Figure 6 shows the number of seeds that appear at a given number of locations for the reference sequence chr1m and query sequence chr2h. As it can be seen from the figure, there are over 10 million seeds that appear in only one location and over 2 million seeds that appear in six locations which is still a significant portion. Hence, without load balancing, i.e., if a thread are statically allocate for a seed, the load imbalance may yield an insufficient performance. Thus, the figure clarifies our motivation behind the load balancing.

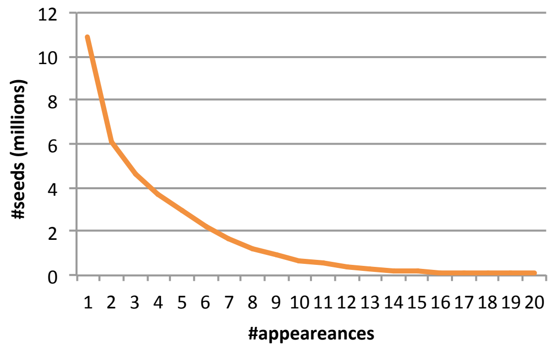


Figure 6. The number of seeds that appear at a given number of locations for the reference sequence chr1m and query sequence chr2h.

As Figure 7 shows, especially for the large sequences (the first five configurations), the proposed load balancing heuristic increases the performance 1.6 to 4.4 times. Furthermore, the speedup increases for smaller L values, i.e., for relatively harder problems. For example, for the configuration with chr1m and chr2h as the reference and query sequences, respectively, and $L = 30$, GPUMEM extracts MEMs in 88.87 seconds without load balancing which in fact is worse than the time essaMEM spends with $\tau = 8$ threads. Yet, with load balancing, GPUMEM becomes 1.6 times faster than essaMEM.

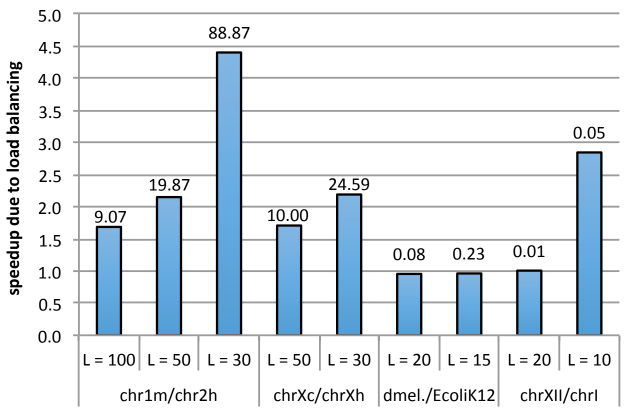


Figure 7. MEM extraction times of GPUMEM without load balancing (in seconds, over the bars) for the nine configurations in our experiments and their ratio to the time with load balancing, i.e., speedup due to load balancing.

V. Conclusion and Future Work

We proposed a novel tool GPUMEM for the maximal exact match extraction problem which, to the best of our knowledge, is the first GPU-based tool. The proposed tool uses a lightweight index structure that is suitable for memory restricted devices and easy to generate with thousands of threads in parallel. Our experiments show that the proposed tool is faster than the state-of-the-art CPU-based tools in our experimental setting. GPUMEM partitions the 2D (reference/query) search space into smaller blocks and solve each block separately in a memory-restricted GPU device.

In future work, we are planning to investigate the variants of the maximal exact match extraction problem such as unique and rare exact match extraction. Furthermore, we will study on novel GPU-based indexing techniques which can increase the performance of the extraction phase. Although we believe that the improvements will be better, we also want to evaluate the performance of GPUMEM with newer GPUs such as Tesla K40.

VI. Acknowledgments

This work was partially supported by the NHI/NCI grant R01CA141090. This publication was made possible by NPRP grant #4-1454-1-233 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors. We also thank to NVIDIA for providing us the K20 card used in the experiments.

References

[1] M. Abouelhoda and S. Seif. Efficient distributed computation of maximal exact matches. In *Recent Advances in the Message Passing Interface*, Lecture Notes in Computer Science, pages 214–223. Springer Berlin Heidelberg, 2012.

[2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. The 9th International Symposium on String Processing and Information Retrieval.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J Mol Biol*, 215:403–410, 1990.

[4] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4-5):327–344, 1994.

[5] J.-H. Choi, H.-G. Cho, and S. Kim. Game: A simple and efficient whole genome alignment method using maximal exact match filtering. *Computational Biology and Chemistry*, 29(3):244 – 253, 2005.

[6] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of whole genomes. *Nucleic Acids Res*, 27:2369–2376, 1999.

[7] D. Edwards and K. Holt. Beginner's guide to comparative bacterial genome analysis using next-generation sequence data. *Microbial Informatics and Experimentation*, 3(1):2, 2013.

[8] F. Fernandes and A. T. Freitas. slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array. *Bioinformatics*, 2013.

[9] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS 2000*, pages 390–. IEEE Computer Society, 2000.

[10] S. Garcia, J. Rodrigues, S. Santos, D. Pratas, V. Afreixo, C. Bastos, P. Ferreira, and A. Pinho. A genomic distance for assembly comparison based on compressed maximal exact matches. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 10(3):793–798, 2013.

[11] Z. Khan, J. S. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence data sets using sparse suffix arrays. *Bioinformatics*, 2009.

[12] 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*, 5(2):R12, 2004.

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

[14] E. Ohlebusch and S. Kurtz. Space efficient computation of rare maximal exact matches between multiple sequences. *Computational Biology*, 15(4):357–377, 2008.

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

[16] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.