Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance

Abdullah Gharaibeh, Matei Ripeanu

{abdullah, matei}@ece.ubc.ca

Abstract—**GPUs offer drastically different performance characteristics compared to traditional multicore architectures. To explore the tradeoffs exposed by this difference, we refactor MUMmer, a widely-used, highly-engineered bioinformatics application which has both CPU- and GPU-based implementations.**

We synthesize our experience as three high-level guidelines to design efficient GPU-based applications. First, minimizing the communication overheads is as important as optimizing the computation. Second, trading-off higher computational complexity for a more compact in-memory representation is a valuable technique to increase overall performance (by enabling higher parallelism levels and reducing transfer overheads). Finally, ensuring that the chosen solution entails low pre- and post-processing overheads is essential to maximize the overall performance gains.

Based on these insights, MUMmerGPU++, our GPU-based design of the MUMmer sequence alignment tool, achieves, on realistic workloads, up to 4x speedup compared to a previous, highly optimized GPU port.

I. INTRODUCTION

High-performance computing (HPC) platforms are gradually shifting towards hybrid architectures. Simply put, hybrid architectures combine high-frequency processors with massively-multicore, yet low-frequency, accelerators. This combination makes perfect sense as applications typically have both sequential parts, run by the fast, high-frequency processor, and parallel parts, run by the accelerators. As argued by Hill et al. [1], compared to homogeneous multicore systems, hybrid architectures offer a better balance between performance and used resources (power and processor area). Examples of such hybrid platforms include IBM's Cell Broadband Engine, AMD's Fusion architecture, Intel's Larrabee chip, and commodity systems that host both traditional CPUs and commodity graphics processing units (GPUs).

Experience with hybrid platforms powered by general purpose GPU (GPGPUs) includes reports of significant speedups compared to current homogeneous multicore systems in the same price range [2]. These reports ignited passionate debate on the limits of GPU-supported acceleration for various classes of applications [3, 4].

Indeed, designing applications to run efficiently on a hybrid, GPU-based platform is a challenging task for multiple reasons:

- First, the GPU processing power is offered in a restricted form of parallelism, known as single-instruction multiple-data (SIMD), which allows for only one instruction to operate on multiple data items at each point in time. Hence SIMD provides lower execution flexibility and requires extracting parallelism at the low-level.

- Second, splitting the computation between the CPU and the GPU requires explicit data transfers between their address spaces over a shared I/O bus. Hence efficiently scheduling data transfers between the two processing units, and finding a low coupling point that limits the volume of data transferred, are required to achieve best performance results.

- Finally, and most relevant to this work, most of the past experience on performance-efficient data structures needs to be carefully reconsidered when porting applications to use GPUs. The reason is that *GPUs offer different computational tradeoffs compared to traditional multicore systems*. On the one hand, GPUs offer one order of magnitude higher peak memory access bandwidth, and one order of magnitude higher peak computational power. On the other hand, current GPUs have limited, often one order of magnitude lower, internal memory space; moreover their computational model results in extra overheads as it relies on transferring data back and forth between the device and the host system's main memory.

This paper advocates the need for a careful space/time tradeoff analysis when designing applications for (or porting applications to) GPU-based hybrid platforms. In particular, we analyze and evaluate these tradeoffs in the context of a well-engineered, widely-used bioinformatics application [5-7] which performs 'read alignments': a memory-intensive operation involving exact string matching for a large number of strings. The tool has both CPU- and GPU-based implementations named MUMmer [5-7] and, MUMmerGPU [8, 9], respectively.

Using a GPU to accelerate the 'read alignment' operation is appealing for two reasons. First, GPUs support higher peak memory bandwidth than traditional systems. This is facilitated by faster memory technology used by GPUs, named GDDR, and by employing a wider memory bus. Second, parallelizing this operation is straightforward since matches can be processed independently and the problem space can be easily partitioned. Therefore, it is not surprising that, after careful optimizations [9] to efficiently use GPU's texture memory and improve data access locality, MUMmerGPU achieves significant speedups compared to its CPU counterpart.

Profiling the latest version of MUMmerGPU, however, reveals that only a relatively low share of the total application runtime is spent computing. Figure 1 shows that more than 50% of the time is spent on data transfers and post-processing.

Our hypothesis is that the culprit for this arguably low use of the GPU is the core data structure (namely the *suffix tree*) used for performance-efficient string matching by both the original, CPU-based tool, MUMmer, and its GPU port. We contend that this data structure is not a good match for GPU implementations: it offers fast matching at the cost of a large memory footprint (which translates to large data transfers and limited parallelism) and relatively complex post-processing.

Thus, the goal of this study is to: first, explore the feasibility of using a different data structure that offers different space/time tradeoffs and, second, to evaluate the effect of this choice on the overall application performance. Note that, to highlight the effect of the choice of the data structure, we focus on the high-level application design, and, throughout our design and implementation effort, we pay little attention to low-level performance optimizations.

The contributions of this work are as follows:

- First, we demonstrate the importance of a careful choice of the data structure used to support a GPGPU application. We show that a data structure that matches well the space/time tradeoffs specific to a GPU can lead to dramatic performance improvements. The direct implication of this observation is that, when porting applications to a hybrid, GPU-supported platform, designers should not only focus on extracting the application parallelism usable in a SIMD model; but, in order to maximize the performance gains, they may need to reconsider the choice of the data structures used.

- Second, we provide MUMmerGPU++, a fully compatible GPU port of the widely used sequence alignment tool MUMmer. Our evaluation, performed on one commodity and one high-end GPU card using realistic workloads, which include large-scale human genome sequencing data, demonstrates that MUMmerGPU++ enables up to 4x speedup compared to MUMmerGPU, the highly optimized GPU port of the original MUMmer sequence alignment tool. As we argue in §V, data analysis has become a major bottleneck in generating new knowledge from genomic data [10]; thus, accelerating sequence alignment, a major step in data analysis, has the potential to alleviate this bottleneck.

- We contrast, in the context of the sequence alignment application, energy consumption for traditional and hybrid (GPU-enabled) systems. We show that, although the energy consumption *rate* of a traditional system is lower, the *total* energy consumption to complete a full sequence alignment workload for the hybrid system is much lower due to its superior performance. While, for our experimental setup, the hybrid system requires only a small (13%) performance improvement to become more energy efficient than a traditional one, MUMmerGPU++ offers much higher speedups thus making the hybrid system over one order of magnitude more energy efficient.

How to read this paper. To make this paper self-contained we present a fair amount of background material. If the reader is familiar with the read alignment problem (§II.A), the data structures to accelerate string matching and their space/time tradeoffs (§II.B), and the GPGPU programming model (§II.C) then he can skip directly to §III, which discusses in detail the effect of space/time tradeoffs and our effort to offload read alignment computation to the GPU. §IV presents a detailed evaluation of our solution, MUMmerGPU++, and compares it with the past approach. §V extends the performance evaluation
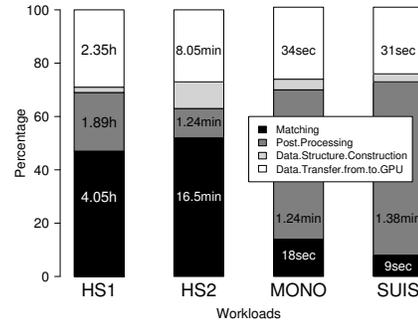

Figure 1: Percentage of time spent in each processing stage using MUMmerGPU for the workloads presented in Table 1, for *config2* (discussed in §II.A.2).

over multiple directions (e.g., energy consumption, ability to harness high-end GPUs) and discusses a number of interrelated design issues. §VI concludes this paper.

II. BACKGROUND

Genome sequencing is the biochemical process of determining the order of nucleotides in a DNA molecule. This is an essential process to gain important information needed for biological and medical studies. New high-throughput sequencing technologies, such as 454 life sciences [11] and Illumina [12], enabled dramatic increase in sequencing rates, while significantly reducing the overall sequencing costs. This advancement enables producing an enormous volume of data (generated at the rate of terabytes per day) which needs to be processed and analyzed, leading, as a result, to insatiable demand for high-performance computing.

This paper focuses on sequence alignment: the operation on genomic data which aims to find all occurrences of one sequence in another sequence, where a sequence is a string composed of some alphabet Σ (e.g., the alphabet set {A,C,G,T} in case of genome sequences). Sequence alignment [13] is widely used in computational biology studies such as gene finding, comparative genomics, phylogenetic analysis and genome assembly. In particular, we focus on a specific, yet important, use case in sequence alignment, called genome read alignment.

A. The Read Alignment Problem

In *read alignment*, a large number of short sequences, (called 'reads') and referred hereafter as the query set, are aligned to a longer genome *reference sequence*. This process is an essential time-intensive operation in comparative genome assembly [14].

1) Formal Problem Definition

The read alignment problem can be formally defined as follows: For each query q in the query set Q, find all *maximal matches* of minimum length l in the reference string S. A *maximal match* is defined as a match of a suffix q_i of query q starting at position i (and referred hereafter as a subquery) to a suffix S_j of the reference string S that is at position j. The match is assumed to be

Table 1: Sample 'read alignment' workloads. For experimental purposes, we use three different minimum-match length values

| Workload / Species	Reference sequence length	# of queries	Sequencing technology (read length)	Minimum-match length
HS1 - Homo sapiens chromosome 2	238,202,930	78,310,972	454 (~200)	Config1: 25, Config2: 50, Config3: 100
HS2 - Homo sapiens chromosome 3	100,537,107	2,622,728	Sanger (~700)	Config1: 50, Config2: 100, Config3: 200
MONO - L. monocytogenes	2,944,528	6,620,471	454 (~120)	Config1: 20, Config2: 40, Config3: 80
SUIS - S. suis	2,007,491	26,592,500	Illumina (~36)	Config1: 15, Config2: 20, Config3: 30
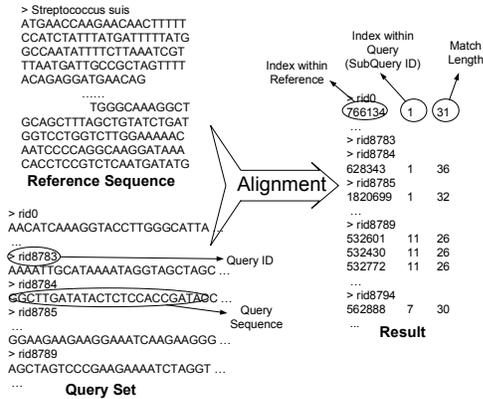
Figure 2: Genome read alignment example.

as long as possible, and not contained in any suffix q_k, with $k < i$.

For example, for a query string "ACACT" and a match length of at least three, the following three subqueries must be searched in the reference string: ACACT, CACT, and ACT. For each subquery, all match occurrences that are at least three characters long must be reported. Figure 2 shows a snapshot of a reference sequence, query set and alignment result.

2) Workload Characteristics

Depending on the species, the length of the genome reference sequence ranges from a few million nucleotides (e.g., for *Streptococcus suis*), to a few billion nucleotides (e.g., for *Homo sapiens*), to a maximum of hundreds of billions nucleotides (e.g., for *Amoeba dubia*). A nucleotide is represented as a character from the alphabet set {A,C,G,T}.

The number of queries ranges from few thousand to hundreds of millions, and the query length ranges from tens to several hundred nucleotides depending on the sequencing technology used. In particular, current high-throughput sequencing technologies, such as Illumina and 454, produce significantly shorter queries (30–200 nucleotides) compared to previous sequencing generations such as sanger (~700 nucleotides).

Table 1 presents a sample of read alignment workloads we fetched from the National Center for Biotechnology Information (NCBI) archive [15], and used to drive the experiments in this paper. The workloads include sequencing data that cover a wide spectrum of usage scenarios. For example, **HS1** is a relatively large scale workload for a *Homo sapiens* that aligns about 78M queries of average length 200 to the genome sequence of the human chromosome #2 which is about 238M nucleotide-long. **MONO** is a smaller scale workload for a *Listeria monocytogenes* species which aligns ~6M queries to a reference genome sequence ~2M nucleotide-long.

Finally, the minimum match length is a user-specified parameter. A short minimum-match length implies a relaxed assumption on what is considered a match, and vice versa. On the one hand, since all the suffixes of each query need to be aligned, a short minimum-match length increases the number of subqueries to be aligned per query, and, at the same time, increases the chance to find matches; therefore the workload becomes larger, and requires more processing time. On the other hand, a longer minimum-match results in reducing the workload demands.

For each workload, we chose three minimum-match length values that represent relaxed (config1), moderate (config2) and conservative (config3) configurations with respect to typical
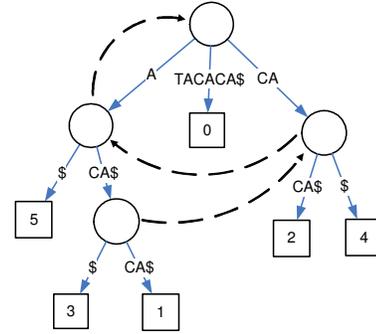
values used in practice [8, 9] (see Table 1).

B. Substring Matching

The core of the read alignment problem is a basic substring matching operation: find a string of length m in another reference string of length n, where $n \gg m$. A naïve approach to this problem is to exhaustively search the reference string. This approach has linear space complexity, $O(n)$; in fact, if a nucleotide is represented using one Byte, the space requirement of this approach is exactly n Bytes. However, the time complexity is daunting: $O(mn)$, especially when considering that matching needs to be done on a large number of queries.

A more time-efficient approach to solve this problem is to pre-process the long reference string into a data structure that allows for efficient search. The rest of this section discusses the two main data structures that have been proposed in the literature: suffix trees [16] and suffix arrays [17].

1) Suffix Tree

A suffix tree (Figure 3) is a trie-like data structure that stores all the suffixes of a given string S (the reference string in our case). Each suffix has exactly one path from the root of the tree to a leaf. The tree has n leaf nodes, corresponding to the n suffixes in S. Further, each edge in the tree is labeled with a substring of S such that the concatenation of the edge-labels from the root to a leaf represents a suffix S_j of S.

Search procedure and its complexity. Searching the suffix tree is done by navigating the tree starting from the root node, matching the characters of the query string with the edge-labels. The search complexity is $O(m)$, where m is the length of the query string. This is an attractive linear-time search solution which does not depend on n, the length of the reference. Also, suffix trees can be augmented with additional pointers, called *suffix links* (shown as dashed arrows between internal nodes in Figure 3), which enable time-efficient maximal-matching (discussed below). Conceptually, a suffix link is an internal pointer from a node with path αw (i.e., the concatenation of edge-labels from the root to the node) to another node with path w, where α is a single character and w is a substring.

Processing the maximal-matches of a query q of length m requires searching the suffix tree for all subqueries q_0 to q_{m-l} (where l is the minimum-match length). This can be done by treating each subquery as a separate query, and performing a separate search operation for each one. However, this approach fails to take advantage of the fact that we are searching for a group of related suffixes. To this end, suffix links allow for exploiting this opportunity: instead of traversing the suffix tree from the root

Figure 3: The suffix tree for the string TACACA. Dashed arrows represent suffix links.

node for each subquery, the matching can be resumed for subquery q_i by following the suffix link of the last matching node of the previous subquery q_{i-1}, hence saving i-1 comparisons for each suffix, and rendering the complexity of matching *all* the subqueries of a query to be O(*m*).

Space complexity. The time efficiency of the suffix tree comes at the cost of additional computational and space overheads to build and store the suffix tree. Although the space complexity grows linearly with the reference sequence length as the tree requires only O(*n*) nodes, in practice the constant factors are high and suffix trees occupy a significant amount of space: between 22.4*n* and 32.7*n* Bytes for DNA sequences [17-19], where *n* is the sequence length. Storing the suffix links will require 4*i* additional Bytes, where *i* is the number of internal nodes. As a result, efforts have been made to reduce the space requirements of the tree, which resulted in reducing the space requirement to 20*n* Bytes in the worst case [19], without considering the suffix links.

Construction. The tree can be constructed in O(*n*) time [16], which in practice becomes negligible when matching a large number of queries. Further, suffix links are a by-product of suffix tree construction, hence no extra pre-processing time is required to produce them, yet they still consume additional space to store.

2) Suffix Array

To address the large space requirements of suffix trees Manber et al. [17] proposed the suffix array, a data structure that enables similar string matching operations yet consuming less space in practice. A suffix array is a sorted array of all the suffixes of *S* in lexicographical order [17] (presented in Table 2 for the same reference string as in Figure 3). The data structure is represented as an array of integers which correspond to the indices of the suffixes in order (column labeled 'suffix array' in Table 2).

Search procedure and its complexity. A naïve search in the suffix array takes O(*m*log *n*) time when supported by a classic binary search: O(*log n*) string comparisons demanded by the binary search, and each string-comparison requires O(*m*) character comparisons. In practice, however, a smart implementation of the binary search that takes advantage of the fact that we are searching related suffixes significantly improves the search time. Manber et al. proved that the *worst case* time complexity can be improved to O(*m* + log *n*) [17] at the expense of increased space usage by associating the suffix array with an extra array of information, namely the longest common prefix (LCP) array: an array that stores the length of the longest common prefix between the suffix stored in the current entry and that stored in the previous array

Table 2: Suffix array for the string TACACA. The suffix and index columns are shown for illustration only (i.e., they do not present in the actual data structure). The *LCP array* represents the longest common prefix between the suffixes in the current and the previous array entry. The *rank array* represents the reverse index of the suffix array and has the same role as the suffix links in suffix trees: it is used to efficiently calculate maximal matches as discussed in §III.C.

Index	Suffix	Suffix Array	LCP Array	Rank Array (Suffix Array $^{-1}$)
0 (smallest)	A	5	0	5
1	ACA	3	1	2
2	ACACA	1	3	4
3	CA	4	0	1
4	CACA	2	2	3
5 (largest)	TACACA	0	0	0

entry. Using the LCP array allows 'priming' the binary search: that is, the search does not start from scratch for each string-comparison. In a nutshell, the results of earlier string-comparison iterations along with the LCP information are used to skip unnecessary comparisons in subsequent iterations.

Space complexity. The suffix array has O(*n*) entries, the same asymptotic space complexity as the suffix tree; in practice, however, it consumes three to five times less space than suffix trees [17, 18]. In particular, if we assume an integer is represented on four Bytes, the array requires exactly *4n* Bytes. The LCP and the rank array (discussed in §III.C) add another 8*n* Bytes.

Construction. The suffix array can be constructed in linear time [20-22]. As with the suffix tree, construction overheads are amortized even for a relatively small number of queries.

C. GPGPU Programming

At the high-level, offloading computation to the GPU is an iterative process of three stages: (*i*) transfer the input data to the GPU's internal memory, (*ii*) launch the processing '*kernel*', a function that, when called, is executed on the GPU, and (*iii*) transfer the output from the GPU's internal memory back to the host's main memory.

This processing model is imposed by the fact that the GPU has no direct access to the host's memory nor to its I/O devices (e.g., disk). Rather, the internal GPU processors can only access the memory module hosted by the GPU card, itself. Consequently, the application has to allocate buffers on the GPU's local memory for both input and output data, and to transfer the data to/from these buffers from/to the host's main memory.

The GPU architecture and programming model have been discussed extensively in several previous publications; hence we refer the reader to [23] for a detailed presentation on this subject.

III. OFFLOADING READ ALIGNMENT

This section discusses the challenges to offload read alignment to the GPU (§III.A), presents MUMmerGPU's approach to read alignment based on suffix trees (§III.B), discusses our design of a suffix array-based tool (§III.C), and the opportunities it enables to gain extra speedups by significantly reducing the data transfers and post-processing overheads (§III.D).

A. Challenges

The efficient use of GPUs to speedup read alignment faces two main challenges:

- *Limited onboard GPU memory.* Current GPU models have one order of magnitude less memory compared to the host's main memory. This limitation may constrain applications to partition the problem space and perform computations in several rounds, hence adding significant data transfer overheads especially for data-intensive applications.

  The space requirement of the read alignment problem is fairly large, especially when considering long sequences such as those of mammalian genomes [24]. For example, the human reference genome spans more than 3 billion DNA nucleotides (i.e., more than 3GB string) which, when processed into a suffix tree or suffix array, would require significantly more space (20x more, i.e., 60GB when using a suffix tree). Moreover, current sequencing projects typically produce more than 10x oversampling of the genome (i.e., the total length of

all queries is 10x the length of the reference sequence) which needs to be aligned against the entire reference genome [25]. As a result, the space requirements of the problem are at least one order of magnitude larger than the size of the onboard memory in current and near-future GPU models (for example, Nvidia has recently announced its new commodity GPU model, GeForce GTX 480, which has 1.5GB of onboard memory; current high-end GPU models like the Quadro FX 5900 have up to 6GB of onboard memory).

- *Limited access to other I/O devices (e.g., disk).* As mentioned before, the GPU has access only to its onboard memory; hence results have to be stored internally then transferred to the host's main memory. As a result, GPU applications with a large output size need to divide the limited onboard memory efficiently between the input and output buffers. This becomes a challenge when the result size cannot be determined in advance for a specific input size, or the maximum result size is too large to be allocated. Addressing this limitation requires a compressed, deterministic representation of the results, which needs to be decompressed on the CPU (or possibly by another round on the GPU), consequently introducing extra overheads.

  In the case of read alignment problem, the output size cannot be determined in advance as the number of alignments for each subquery is not known beforehand. Moreover, the maximum result size is O($mn|Q|$), which is infeasible to allocate.

B. A Previous Effort: MUMmerGPU

Delcher et al. [5, 6] implemented MUMmer, a widely used tool that performs read alignments on the CPU using suffix trees. The tool has also been significantly improved in terms of performance and space efficiency by Kurtz et al. [7]. Schatz et al. developed [8] then optimized [9] a GPU version of the program, called MUMmerGPU, which also uses suffix trees. To address the space challenges of the problem (i.e., the long reference sequence, the large number of queries, the unpredictable result size, and the limited onboard GPU memory), MUMmerGPU divides the computation into smaller-sized sub-computations that fit on the GPU onboard memory. This is done by (*i*) dividing the long reference string into shorter overlapping *segments*, (*ii*) dividing the query set into smaller sized subsets, and (*iii*) reporting a "compressed" representation of the results to the host's memory. Figure 4 presents the high-level GPU offloading algorithm employed by MUMmerGPU.

MUMmerGPU constructs a suffix tree for each segment (a partition of the reference string), and aligns each query subset to all trees in rounds. Conceptually, a "round" is a four-stage process (for a more detailed discussion, we refer the reader to [8, 9]):

- *Copy in.* The query subset and the suffix tree of the segment are transferred to the GPU.

```
structs = PreprocessReference(reference)
subsets = DivideQuerys(queries)

foreach subset in subsets do {
   results = NULL
   CopyIn(subset)

   foreach struct in structs do {
      CopyIn(struct)
      LaunchMatchKernel(subset, struct)
      CopyOut(results) /* append result */
   }

   Postprocess(results)
}
```

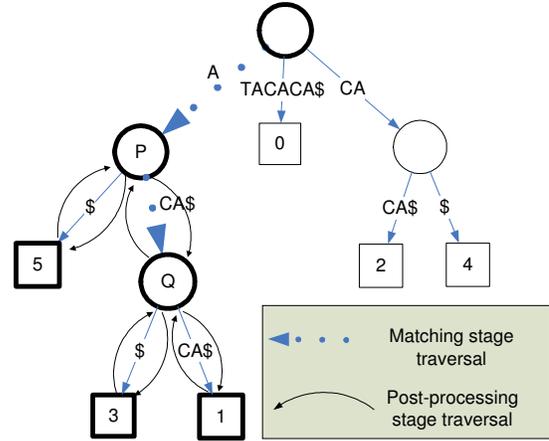**Figure 4: High-level GPU offloading algorithm**

Figure 5: Alignment of query ACACT to reference TACACA for a minimum-match length of one. The figure demonstrates the alignment for only the first subquery (i.e., the string ACACT, itself). The dotted path is traversed in the matching stage. Node Q, and the corresponding maximum match length of 4, are reported as the result of the traversal in the matching stage. The post-processing stage produces the final output through a depth-first traversal starting from node P. The output includes three alignments: at position 5 with length 1, at position 3 with length 3 and at position 1 with length 4.

- *Matching.* The queries of a query subset are aligned to the tree in parallel on the GPU. All subqueries of a query are processed by a single GPU thread in order to take advantage of suffix links. To make the result size predictable, the match kernel does not report all the matches of each subquery (as discussed previously, a subquery could have one or more matches; however, the number of matches is not known in advance). Instead, the match kernel reports only the longest match of each subquery (node "Q" in Figure 5). This is done by matching the characters of the subquery string with the edge-labels until a mismatch or the end of the subquery is reached.

- *Copy out.* The results are transferred back to main memory.

- *Post-processing.* The results of the match kernel are "decompressed" to find the other matches of each subquery. This is done as follows (Figure 5 presents an example). First, starting from node Q that corresponds to the longest match for a subquey, the algorithm traverses back to the node at which the match length equals the minimum-match length *l* (labeled P in Figure 5). Intuitively, P is the lowest common ancestor of the leaves that represent all subquery matches. Second, the algorithm performs a depth-first traversal to report all the leaves of the subtree rooted at P as the final result (i.e., the indices in the reference string where the subquery occurs).

C. MUMmerGPU++

Schatz and his group report that MUMmerGPU achieves significant speedups compared to the original CPU-based MUMmer program [8]. A closer look at the match between suffix tree-based search and the GPU characteristics prompted us to investigate whether a suffix array implementation can enable better utilization of the GPU. This section presents the suffix array-based algorithms used by MUMmerGPU++ while the

```
/* Assumes SA, LCP and l global variables */
procedure Match(q, qlen) {
    i = 0
    while i ≤ qlen - l do {
        (si, ml) = BinarySearch(q_i)
        RecordResult(q_i, si, ml)
        i = i + 1
        while si != NULL and i ≤ qlen - l do {
    /* phase 1: cut the search space */
            i = i + 1
            s = ml - 1
            si = Rank[SA[si] + 1]
            j = SA[si] + s
            (r, ml) = Comp(S_j, q_{i+s})
    /* phase 2: find the longest */
            if r > 0 then {
                (si, ml) = ScanUp(s+ml, q_i)
            } else {
                (si, ml) = ScanDown(s+ml, si, q_i)
            }
            RecordResult(q_i, si, ml)
            i = i + 1
        }
    }
}
procedure ScanUp(s, si, q_i) {
    r = 1
    while LCP[si] > s and r > 0 do {
        si = si - 1
        j = SA[si] + s
        (r, ml) = Comp(S_j, q_{i+s})
        s = s + ml
    }
    return (si, s)
}
```

Figure 6: Pseudo-code of the core matching algorithm of MUMmerGPU++. The procedure "Match" is executed for each query by a dedicated GPU thread. The following is a summary of the variables names used: _i_=subquery index, _l_=minimum match length, _ml_=match length, _s_=skip (processed characters), _si_=suffix index. The procedure "Comp" evaluates which string is greater lexicographically and returns the maximum match length. Finally, the procedure "ScanDown" is similar to "ScanUP" but examines the entries in the other direction by incrementing the suffix index _si_.

following section estimates analytically the potential performance gains brought by this data structure.

At the high level, MUMmerGPU++ follows the same structure as MUMmerGPU (described in Figure 4). However, the core of MUMerGPU++ is significantly different as we replace the core data structure, the suffix tree, with a suffix array. This change entails completely different matching and post-processing algorithms, which we describe in the rest of this section.

Matching. Similar to MUMmerGPU, queries are searched in the suffix array in parallel, and all subqueries of a query are processed sequentially by a single GPU thread. For each subquery, the match kernel reports the index in the suffix array corresponding to the longest match in the reference.

The matching algorithm processes a query q as follows: the first subquery q_0 is matched via a binary search on the suffix array, which, as discussed in §II.B.2), has $O(m + \log n)$ worst case complexity, where m is the query length and n is the reference string length. To process the next subquery and avoid processing the characters already processed by the previous subquery, we use a two-phase procedure (pseudo-code presented in Figure 6):

- The first phase uses the result of the previous subquery to reduce the search space in the suffix array. This is done by

```
/* Assumes SA, LCP and l global variables */
procedure PrintSubQueryAlignments(i, si, ml){
    /* print the longest one */
    PRINT(SA[si], i, ml)
    /* Scan up */
    v = si
    m = ml
    while v > 0 and m ≥ l do {
    /* the lcp could be longer than the
       match length, hence the minimum */
        m = MIN(m, LCP[v])
        v = v - 1
        PRINT(SA[v], i, m)
    }
    /* Scan down */
    v = si + 1
    m = MIN(ml, LCP[v])
    while v < reflen and m ≥ l do {
        PRINT(SA[si], i, ml)
        v = v + 1
        m = MIN(m, LCP[si])
    }
}
```

Figure 7: Pseudo-code of the core post-processing procedure. This procedure is invoked for each subquery in each query to decompress the result of the matching stage.

combining the suffix array with another one called the *rank array*: the reverse index of the suffix array (see Table 2).

For example, let S_j be the reference suffix that matched x characters of subquery q_i, where $x >= l$, also let k be the rank of S_j in the suffix array (i.e., SuffixArray[k] = j and Rank[j] = k); then the subquery q_{i+1} matches x - 1 characters of the reference suffix S_{j+1}, and Rank[S_{j+1}] is the corresponding suffix array index. Conceptually, the Rank array has the same role as the suffix links in suffix trees.

- The second phase searches for the longest match by sequentially comparing the subquery with the suffixes adjacent to the one produced by the first phase. The LCP array is used to avoid comparing a character more than once.

Note that if a subquery does not have a match in the reference string, the search for the next subquery falls back to the binary search procedure. Hence, the efficiency of this approach is related to the characteristics of the workload: the larger the number of matching subqueries, the lower the number of times the algorithm searches the whole array.

We anticipate that this approach is efficient for the read alignment problem since generally the queries are aligned to a reference genome of the same species; hence the percentage of positive matches is relatively high.

Post-processing. The result reported by the match stage represents the longest match occurrence for each subquery. Since the suffix array is ordered lexicographically, the other occurrences are adjacent: above and under the result reported by the match phase. Getting the other occurrences, and their maximum match length, is done via a simple sequential scan on the LCP array. The algorithm is presented in Figure 7.

D. Evaluating the Opportunity: A Detailed Analysis of Space/Time Tradeoffs

This section uses simple complexity analysis to shed light on the effect of using suffix arrays instead of suffix trees on the running time of each of the matching, data transfer, and post-processing stages. In brief this section argues that even though suffix arrays will not enable a faster matching stage, they will enable significantly lower data transfer volumes and faster post-

processing. These gains can be significant as these two stages consume a large share of MUMmerGPU processing time (between 50% and 93% depending on the workload as seen in Figure 1). The evaluation using real workloads presented in §IV supports these conclusions.

1) The Matching Stage

Suffix trees and suffix arrays provide different trade-offs in search and space complexity which can be summarized as follows: on the one hand, suffix trees support $O(m)$ search complexity, while suffix arrays support $O(m + \log n)$ to align a string of length m to a reference string of length n; on the other hand, although their asymptotic space complexity is similar, in practice suffix arrays are 3-5x more space efficient than suffix trees.

As mentioned before, tackling the constraints imposed by limited memory requires dividing the large query set into smaller subsets, and the long reference sequence into shorter segments.

To compare the time complexity of the matching stage for suffix trees and suffix arrays we use the following notations: let k be the number of query subsets and c_d be the number of segments the reference is divided into when using data structure d. Also, let t_d be the time complexity of matching a single query on the GPU. Finally, let α be the ratio between the number of queries in a query subset and the number of SIMD processors in the GPU. Note that, by assuming that α does not depend on the data structure used, we implicitly assume that the size of a query subset is the same for both the array- and tree- based solutions, and that the space savings achieved in the suffix array-based solution will be used to increase the segment size (i.e., reduce the number of segments c_d).

Since processing all queries requires matching all query subsets to all reference segments, the time complexity of matching all queries using data structure d can be expressed as:

$$T_d = kc_d t_d \alpha$$

Suffix tree-based tool. As discussed in §II.B.1), suffix links enable $O(m)$ search time for all subqueries (suffixes) of a single query. As a result, the time complexity to search a query on the GPU using suffix trees can be expressed as: $t_{tree} = O(m)$. Thus, the time to process the query on all segments using suffix trees can be expressed as:

$$T_{tree} = kc_{tree}\alpha O(m)$$

Suffix array-based tool. In the case of suffix arrays, $t_{array} = O((m + \log(n/c_{array}))/r_{array})$, where r_{array} is the efficiency of calculating the subqueries of a query. Note that r_{array} is less than or equal to one: the value is close to one for workloads with high similarity with the reference, and lower values for workloads with lower similarity. Therefore, the overall time complexity when using suffix arrays:

$$T_{array} = kc_{array}\alpha O((m + \log(n/c_{array}))/r_{array})$$

Speedup. The speedup for the matching stage is:

$$Speedup = \frac{T_{tree}}{T_{array}} = \frac{c_{tree}}{c_{array}} \times \frac{O(m)}{O((m + \log(n/c_{array}))/r_{array})}$$

Since the search procedures for both suffix array and suffix tree exhibit similar behavior: excessive memory access and byte-to-byte comparisons, we assume that the constants in the asymptotic

bound of the search complexity for the suffix array and the suffix tree are close, hence the speedup ratio becomes:

$$Speedup = \frac{c_{tree}}{c_{array}} \times r_{array} \times \frac{m}{m + \log(n/c_{array})}$$

We analyze the three terms that influence the speedup in the formula above. First, from a practical view point, the query length m ranges from 35 to 700 depending on the sequencing technology used; while a reference segment length is up to hundreds of millions of nucleotides (leading to sizes in the order of gigabytes limited by the available memory on the GPU), hence the term $\log(n/c_{array})$ ranges from 20 to up to 30. As a result, the ratio $\frac{m}{m + \log(n/c_{array})}$ is practically between ½, for short queries (small values of m), and one, for long queries. Second, suffix arrays are more space efficient than suffix trees, with a space ratio c_{tree}/c_{array} greater than one, typically three. Finally, as mentioned before, the value r_{array} is less than one, and depends on the workload characteristics.

Summary. The main factors that affect the speedup ratio are: (*i*) the space ratio, which is typically three (*ii*) the query to segment length ratio, which is typically between ½ and 1, and (*iii*) the efficiency of calculating maximal matches in suffix arrays, which depends on the workload. In conclusion, a value of r_{array} larger than 50% makes the running time of the matching phase of a suffix array-based tool comparable with that of a suffix tree-based one, which we anticipate to be the case in realistic workloads as the query-set is aligned to a related reference sequence.

2) The Post-processing Stage

The post-processing stage decompresses the result of the matching stage, and writes the final results to the output file.

Suffix tree-based tool. In the MUMmerGPU case, the matching stage produces a single match for each subquery. Decompressing this into the final result is done via a depth-first traversal for each subquery as discussed in §III.B. This is an expensive pointer chasing procedure, especially when considering typical workloads with millions of queries.

To accelerate this stage, MUMmerGPU performs the decompression on the GPU using a second kernel. Therefore, the post-processing stage is executed as a three-stage GPU offloading process itself: (*i*) copy-in the information required to facilitate post-processing, (*ii*) launch the post-processing kernel which determines the matches for each subquery in parallel and (*iii*) copy-out the final results from the GPU. Note that, due to the same reasons related to GPU memory limitations and the massive output size, offloading the post-processing stage is also done in rounds on the GPU. Finally, once transferred to main memory from the GPU, the results are written to the output file.

Two issues related to the above described GPU offloading process are worth mentioning. First, as mentioned before, it is essential to know the result size of a GPU kernel launch. Hence, in this case, the algorithm needs to know the number of matches for each subquery. MUMmerGPU addresses this is by storing additional information in the suffix tree: each node in the tree stores the number of leaves of the subtree rooted at that node. The post-processing stage is then performed in two phases: the first phase is processed on the CPU wherein, for each subquery, the algorithm traverses back to the node at which the match length equals the

minimum-match length (labeled node P in Figure 5). The number of leaves stored in node P is in fact the number of matches for that subquery, and is used to allocate the required result space on the GPU. The second phase is performed on the GPU where the algorithm determines the matches through a depth-first traversal for each subqeury.

Second, MUMmerGPU designers adopted a stackless depth-first traversal algorithm as, on the GPU, a stackless tree traversal has been shown to be significantly more efficient than an approach that maintains a stack [26]. However, this improvement comes at the cost of, again, storing additional information in the tree: each node in the tree has to store a pointer to its parent node to facilitate this approach.

MUMmerGPU implementers report that offloading the post-processing stage to the GPU enabled a 4x speedup of this stage compared to performing it on the CPU [9]. However, as demonstrated in Figure 1, this stage is still time consuming: it occupies more than 20% of the total processing time. Note that this percentage represents only the post-processing GPU kernel time (i.e., copy-in and copy-out are considered as part of the data transfer overhead discussed in the next section) and writing the final result to the output file.

Suffix array-based tool. MUMmerGPU++ design places the entire post-processing stage on the CPU. As described in §III.C, the matching stage produces a suffix array entry index for each matching subquery. The *LCP* array is then used to determine all other alignments by directly scanning (practically just writing the results to the output file) the entries above and below the reported index with a minimum longest common prefix of *l*.

Summary. On the one hand, a suffix tree-based alignment tool requires costly additional traversal steps in the post-processing stage. MUMmerGPU offloads this stage to the GPU as a second processing round which, by itself, requires a post-processing phase that writes the final results to the output file. On the other hand, a suffix array based tool requires only a simple sequential scan to post-process the results. Hence, we expect the later approach to enable significant time savings for the post-processing stage.

3) The Data Transfer Stage

The GPU is connected to the host via an I/O bus. For a data-intensive application, data transfers represent a significant overhead. As Figure 1 shows, in the case of MUMmerGPU, this stage can take more than 20% of the total execution time.

The main advantage of suffix arrays over suffix trees is space efficiency. A suffix array typically enables three times better space efficiency compared to its suffix tree counterpart. As discussed previously, this space saving enables a suffix array-based alignment tool to divide the long reference sequence into a smaller number of segments, thus reducing the number of GPU execution rounds and the data transfer overhead associated with moving the query set to the GPU.

Additionally, we note that offloading the post-processing stage to the GPU in the suffix tree-based approach entails extra data transfers, which we anticipate to be relatively significant especially when the number of positive matches is large.

IV. EVALUATION

This section presents MUMmerGPU++ performance evaluation: it discusses the experimental setup (§IV.A), presents an evaluation of the speedup delivered by MUMmerGPU++ compared to the most recent version of MUMmerGPU (§IV.B), and investigates the factors that influence the observed performance and the effect of each processing stage on the total execution time (§IV.C).

A. Experimental Setup

The machine used to conduct the experiments has the following characteristics: Intel Core 2 Quad CPU (Q6700) clocked at 2.66 GHz per core, 8GB of host memory, an NVIDIA GeForce 9800GX2 GPU: a dual-GPU card with 128 cores clocked at 1500MHz for each GPU, 1GB of memory. The card is connected to the host via a PCI Express 2.0 x16 bus. (Note that we use only one of the two GPUs on the card). Both MUMmerGPU and MUMmerGPU++ are implemented using CUDA.

The evaluation was done under the real sequencing workloads introduced in Table 1. Unless otherwise mentioned, *config2* (see Table 1) is used as the default configuration for the minimum-match length in the experiments.

It is important to note that our implementation focuses on achieving a good match between the core data structure used and the GPU characteristics. To this end, our implementation is a 'common sense' one that does not aggressively optimize for caching, optimal use of shared memories, or coalesced data access – to enumerate only a few of the optimizations often used.

As a baseline for comparison we use an optimized, recent version (v2.0) of the suffix tree-based MUMmerGPU. This version allows for seven data layout alternatives which determine: first, on which GPU memory type (i.e., global, texture, and constant memory) different parts of the input data (i.e., the reference string, suffix tree, and queries) are placed; and, second, how the suffix tree is stored in memory to enable maximum data access locality to improve cache hit rate when placed in texture memory. For MUMmerGPU (v2.0), these choices resulted in a total of 128 different configuration combinations which affect the performance of the matching and post-processing stages. In their extensive analysis, Trapnell et al. [9] illustrated that the performance of different configuration combinations is sensitive to the workload. However, they concluded that a single configuration provides reasonably good performance across all workloads. This configuration uses "a reordered one-dimensional texture for the suffix tree, global linear memory for the queries and reference", and we use it to configure MUMmerGPU in all our experiments. For a detailed discussion on these configurations, we refer the reader to [9].

As discussed before, due to the limited GPU onboard memory space, the workload is divided into smaller chunks by dividing the reference string into segments, and the query set into subsets processed in rounds. This raises the question on how to divide the onboard memory space between the queries and the reference in each round. Both MUMmerGPU and MUMmerGPU++ follow the same policy: maximize the segment size, while leaving space to accommodate enough queries to feed all cores on the device and extract maximum parallelism. Maximizing the segment size results in reducing the number of segments; this proportionally reduces the matching time as each query is processed fewer times.

For all experiments we exclude the time spent reading queries from the disk as this overhead is the same regardless of the used data structure and lies outside our optimization space. We note that the disk I/O overhead represents 10% to 15% of the total MUMmerGPU execution time for the workloads we used.
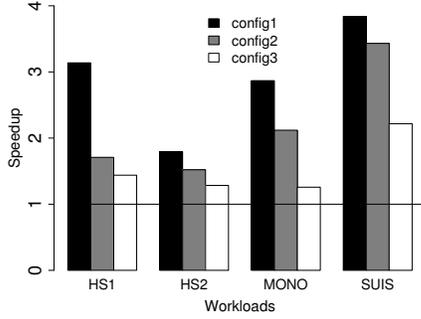
Figure 8: MUMmerGPU++ speedup compared to MUMmerGPU.

Each experiment was run several times, and the execution time was stable in all experiments; hence, we plot only averages (the variations in performance were too small to be visible on the graphs as 95% confidence intervals). Finally, we validated the correctness of our implementation by comparing its output with the one produced by MUMmerGPU over a large number of input sets.

B. Overall Speedup

Figure 8 presents the speedup achieved by MUMmerGPU++ compared to MUMmerGPU for all configurations and workloads presented in Table 1.

While the speedup varies with the workload, MUMmerGPU++ performs better for all workloads: it delivers between 1.25x and 3.83x speedup compared to MUMmerGPU. This significant performance gain is achieved by a better match between the data structure used and the GPU's characteristics. MUMmerGPU++ achieves between 1.52x to 3.43x speedup for what we estimate is the most frequently used configuration (*config2*). The speedup is lower (between 1.25x and 2.21x) for configurations with a longer minimum-match length (*config3*). This is because increasing the minimum-match length decreases the probability of finding matches, hence, as discussed previously, decreases the efficiency of subquery processing when using suffix arrays (represented by r_{array} in §III.D.1), and hurts the performance of the matching stage in MUMmerGPU++. Finally, as expected for a short minimum-match length (*config1*), MUMmerGPU++ offers the best speedup: from 1.7x up to 3.83x.

C. Dissecting the Overheads

To validate our analysis in §III.D, better understand the source of the performance gains observed, and explore the opportunity for further performance tuning, this section explores the absolute and relative time spent in each processing stage.

Figure 9 compares the absolute time spent in each of the processing stages by both MUMmerGPU++ and MUMmerGPU for the largest workload: **HS1**. We note the following:

First, as discussed in §III.D.1), although the suffix tree-based tool, MUMmerGPU, has better asymptotic time complexity per query; MUMmerGPU++, the suffix array-based tool, achieves almost equal overall performance because it is more memory efficient and, as a result, requires a fewer matching rounds on the GPU when all queries are considered.

Second, although the post-processing stage in MUMmerGPU is performed on the GPU, the time spent in this stage is reduced by more than a factor of three by MUMmerGPU++, where it is
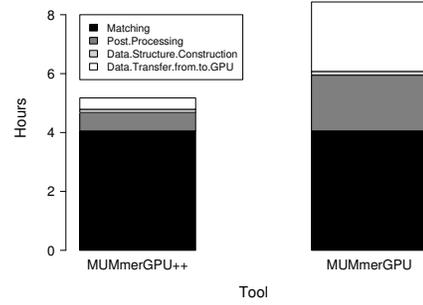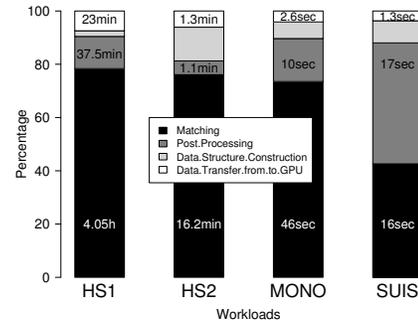

Figure 9: Absolute time spent in each processing stage for workload HS1 for both MUMmerGPU++ and MUMmerGPU (for the default configuration *config2*).

performed on a single CPU core. This translates to 17% overall speedup improvement, hence supporting our argument in §III.D.2).

Third, the experiment validates our insight in §III.D.3) that a suffix array-based tool, like MUMmerGPU++, significantly reduces the data transfer effort from/to the GPU: the total time spent transferring data is reduced by a factor of seven or more, which translates to more than 31% overall speedup improvement.

Finally, for both tools, the time spent in the construction stage is almost negligible compared to other stages.

Figure 10 demonstrates the proportion of total processing time that corresponds to each processing stage for MUMmerGPU++ for all workloads. Compared to Figure 1, which presents similar data for MUMmerGPU, MUMmerGPU++ significantly changes the distribution of processing effort across stages. It significantly reduces the share of post-processing and data-transfer stages, and increases the share of the matching stage.

Figure 10: Percentage of total execution time spent in each processing stage for MUMmerGPU++. The numbers on the bars show the absolute time spent in each stage.

This is important from two perspectives. First, we expect that the tools will be executed on multi-GPU systems. From this perspective, the intense data-transfers employed by MUMmerGPU make the PCI bus a bottleneck and limit the feasibility of using multiple GPUs on the same host. MUMmerGPU++ reduces the I/O overhead (by a factor of 6x-12x in our experiments) and thus eliminates the shared communication (PCI bus) as a potential scalability bottleneck. Second, from a performance optimization perspective, the fact that the compute (matching) stage now takes 75%-80% of the time for the large workloads, including the human genome, allow focusing the performance optimizations on this stage only.

V. Discussion

This section discusses several interrelated questions.

1)   Are the speedups offered by MUMmerGPU++ significant?
DNA sequencing technologies have taken major steps towards commoditization [27]. Moreover, sequencing rates have drastically increased: almost 100 billion nucleotides per day per machine, which is 50,000 times faster than ten years ago [27]. In fact, this considerable improvement encouraged large scale sequencing projects. For example, the 1,000 Genomes project aims to sequence 1,000 human genomes, and will produce more than six trillion nucleotides of data [28]. The widespread availability of sequencing machines, and the associated dramatic increase in daily sequencing rates, must be supported by sequence analysis tools, such as read alignment tools. Indeed, McPherson [10] argues that there is an increasingly growing gap between sequence generation and processing, and that the bottleneck in the ability to generate new knowledge has, in fact, moved from sequencing to the data analysis pipeline, especially for individual investigators and small-scale research laboratories. Accordingly, any performance improvement to these tools translates to saving thousands of much-needed computational hours.

Hence, the speedup offered by MUMmerGPU++ has significant practical implications, especially given the fact that these improvements were achieved only by a better software design, and on commodity hardware.

2)   Is it fair to use MUMmerGPU as a baseline to evaluate the advantages of the suffix array-based approach? Otherwise said, is it possible that the speedup offered by MUMmerGPU++ is simply due to a better optimized GPU implementation and not to the choice of a data-structure that inherently offers a better fit to for the computing platform at hand?
We have three arguments to support our choice of MUMmerGPU as the reference for a suffix tree-based tool. First, our analysis of the opportunities a suffix array-based implementation offers (§III.D.1) is solely based on the characteristics of the core data structure, and is agnostic to the detailed GPU implementation of the tool. Second, MUMmerGPU is a well optimized GPU-based tool. The tool's authors exhaustively examined 128 data layout configurations to select the configuration which delivers the best overall performance. The results were presented in two previous publications [8, 9]. Finally, as mentioned in (§IV.A) we have not specifically optimized the MUMmerGPU++: apart from placing the reference string in texture memory, the kernel places all input and output data in global memory, it does not employ the shared memory available on each multiprocessor and does not try to improve memory throughput by coalescing memory accesses.

3)   Can the data transfer overheads be hidden by overlapping the transfers with the GPU kernel execution?
No, especially for large-scale workloads. The reason is that the computation on the GPU requires a set of input/output buffers. Facilitating communication-computation overlap requires double buffering for the input and output (such that the GPU computes on one set of buffers while the transfers are concurrently performed to/from the others). This entails allocating two sets of input/output buffers on a scarce resource: GPU's onboard memory.

To further investigate this opportunity, we ran an experiment (for both MUMmerGPU and MUMmerGPU++) in which the tool assumed half of the memory available on the device to simulate a double buffering condition. The results demonstrated that the increase in the time spent in the matching stage was larger than the total time spent transferring data from/to the GPU (and could potentially be hidden by the overlapping technique mentioned). Hence, for this application, overlapping would actually hurt the performance. Thus we believe that the opportunity to overlap data transfers with kernel execution has no practical value in this case.

4)   How does MUMmerGPU++ perform in terms of energy consumption compared to its counterparts?
We measured the total energy consumption of the test machine when running the three tools. Note that while the CPU-based version, MUMmer, is widely used and has undergone numerous optimizations [5-7], we have simply used an out-of-the box version of this single-threaded code.

The energy is measured at the wall socket for the execution of an entire workload, hence taking into account the energy usage of the whole compute node (i.e., including the host CPU, I/O, etc.). Two observations are worth noting: first, for all three tools, the energy consumed was linearly proportional to the computation time of the tool; therefore, we show the results (Table 3) for one workload, **HS2**, as all other workloads exhibited similar behavior. Second, both GPU-based tools, MUMmerGPU and MUMmerGPU++, consume energy at the same rate (200 Watt), while MUMmer, the CPU-based tool consumes energy at a lower rate (178 Watt).

Although the CPU-based tool uses energy at a lower rate, the significant runtime reduction achieved by the GPU-based tools renders their total consumed energy much lower than the CPU-based one. For the same reason, MUMmerGPU++ consumes 40% less energy than MUMmerGPU.

Finally, it is important to note that the ratio of the energy consumption *rate* between the hybrid architecture that includes the GPU and the traditional architecture that only includes a CPU reveals that, for our setup, the application running on the hybrid architecture has to offer only 13% better performance to be more energy efficient.

Table 3: Total energy consumption of the test machine in kWh (Killowatt-Hour) for workload HS2. The running time and the energy consumption rate in watts are also shown.

Tool	kWh	Running time (minutes)	Watt
MUMmerGPU++	**0.07**	21	200
MUMmerGPU	**0.12**	36	200
MUMmer	**0.76**	256	178

5)   How do the GPU-based solutions compare when using a high-end GPU model?
We use a recent high-end GPU model: Tesla C1060. Unlike the GeForce series, which targets (gaming) workstations, Tesla targets high-performance computing applications. Compared to GeForce 9800 GX2, Tesla C1060 is more power efficient, has a 4x larger onboard memory (4GB) and 240 cores @ 1.5GHz. Our goal is to evaluate the performance and the portability of MUMmerGPU++ on this new device.

Figure 11 (left) shows that MUMmerGPU++ is at least two times faster than MUMmerGPU. Compared to the speedups achieved over MUMmerGPU on the commodity card (GeForce 9800GX2), MUMmerGPU++ offers significantly better speedups for the larger workloads, **HS1** and **HS2**, while for the relatively small-
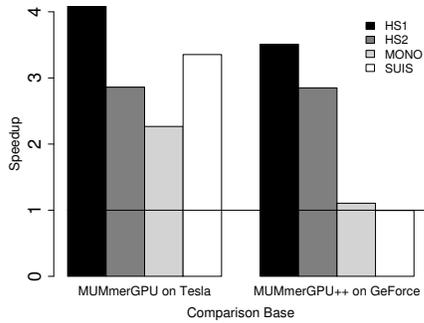
Figure 11: Speedup offered by MUMmerGPU++ when running on a high-end GPU (Tesla C1060) compared to: MUMmerGPU running on the same card (left); and to MUMmerGPU++ when running on the commodity GeForce 9800 (right).

scale workloads, **MONO** and **SUIS,** the speedup achieved is almost the same.

A closer look at the MUMmerGPU code explains this result: to limit the space consumed by the suffix tree, MUMmerGPU designers assumed that the maximum tree size to be 16M nodes, which puts a limit on the maximum reference segment length. This assumption was made to reduce the length of the indices used to access the 2D texture memory, where the tree is placed, to 12-bits in each dimension. This enabled reducing the total size of the suffix tree significantly, while, at the same time, still using the entire memory space low-end cards, such as the GeForce, offer. Our tool, MUMmerGPU++, does not make such assumptions and uses full 32-bit indices for the suffix array; thus it does not implicitly limit its size.

In the case of **MONO** and **SUIS**, the reference strings are short, and do not require segmentation, thus the size limitations mentioned above do not play any role, and the speedup achieved by MUMmerGPU++ over MUMmerGPU is the same as in the previous experiments on GeForce (see Figure 8 for comparison). For the **HS1** and **HS2** workloads which have long reference strings, MUMmerGPU++ is able to use longer reference segments than MUMmerGPU, thus to perform fewer computation rounds and consequently to obtain additional speedup.

Finally, the speedups obtained by MUMmerGPU++ when running on the high-end Tesla card compared to running on the commodity GeForce card (Figure 11, right), support the choice of our policy (discussed in §IV.A) to divide the GPU memory between the reference and the queries, which revolves around maximizing the segment length. As argued above, the **MONO** and **SUIS** have short sequences, hence the extra space offered by the high-end card is used to increase the size of the query subset; however, this does not translate to significant speedups as the GPU memory bandwidth is already saturated. In contrast, on the large-scale workloads, **HS1** and **HS2,** the performance improves significantly due to the opportunity to increase the segment length.

VI. CONCLUSIONS

GPUs have drastically different performance characteristics compared to traditional multicore architectures: up to two orders of magnitude higher peak memory access bandwidth, one order of magnitude higher peak computational power per Byte of memory, yet one order of magnitude lower internal memory space.

We argue that these differences make reconsidering the choice of the data structures used a necessary step when porting applications to hybrid, GPU-supported platforms.

Our experience with MUMmerGPU++, a fully compatible GPU port of the widely used sequence alignment tool MUMmer, supports this conclusion. Our evaluation, performed on a two commodity GPU cards using realistic workloads, which include large-scale human genome sequencing data, demonstrates that MUMmerGPU++ enables up to 4x speedup compared to MUMmerGPU, a highly optimized GPU port that uses, however, the same data structure as the original CPU-based implementation.

We synthesize our experience with porting MUMmer as three guidelines to design efficient GPU-based applications. First, a solution that supports minimum computational overhead does not necessarily enable maximum overall performance: a better optimization point is one that maintains a balance between communication and computation overheads. Second, GPUs' high computational power per Byte of memory compared to traditional multiprocessor architectures, makes trading-off additional per thread processing time for a more compact in-memory data representation an attractive technique to increase overall performance (by enabling higher parallelism levels and reducing data transfer overheads). Finally, ensuring that the chosen GPU-offloaded part of the application entails low pre- and post-processing overheads is essential to maximize the overall performance gains.

REFERENCES

[1] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," Computer, vol. 41, pp. 33-38, 2008.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in Computer Graphics Forum, 2007, pp. 80-113.

[3] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty and P. Hammarlund, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in Proceedings of the 37th Annual International Symposium on Computer Architecture, 2010, pp. 451-460.

[4] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. E. Guney and A. Shringarpure, "On the limits of GPU acceleration," in Proc. of Workshop on Hot Topics in Parallelism, 2010.

[5] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White and S. L. Salzberg, "Alignment of whole genomes," Nucleic Acids Res., vol. 27, pp. 2369, 1999.

[6] A. L. Delcher, A. Phillippy, J. Carlton and S. L. Salzberg, "Fast algorithms for large-scale genome alignment and comparison," Nucleic Acids Res., vol. 30, pp. 2478, 2002.

[7] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu and S. Salzberg, "Versatile and open software for comparing large genomes," Genome Biol., vol. 5, pp. R12, 2004.

[8] M. C. Schatz, C. Trapnell, A. L. Delcher and A. Varshney, "High-throughput sequence alignment using Graphics Processing Units," BMC Bioinformatics, vol. 8, pp. 474, Dec 10, 2007.

[9] C. Trapnell and M. C. Schatz, "Optimizing data intensive GPGPU computations for DNA sequence alignment," Parallel Computing, 2009.

[10] J. D. McPherson, "Next-generation gap," Nature Methods, vol. 6, pp. S2-S5, 2009.

[11] 454 Life Sciences, "http://454.com", 2010 .

[12] Illumina, "http://www.illumina.com", 2010.

[13] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," Briefings in Bioinformatics, 2010.

[14] M. Pop, A. Phillippy, A. L. Delcher and S. L. Salzberg, "Comparative genome assembly," Briefings in Bioinformatics, vol. 5, pp. 237, 2004.

[15] NCBI Trace Archive, "http://www.ncbi.nlm.nih.gov/Traces", 2010.

[16] P. Weiner, "Linear pattern matching algorithms," in Proceedings of the 14th Annual Symposium on Switching and Automata Theory, 1973.

[17] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 319-327.

[18] M. I. Abouelhoda, S. Kurtz and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," Journal of Discrete Algorithms, vol. 2, pp. 53-86, 2004.

[19] S. Kurtz, "Reducing the space requirement of suffix trees," Software: Practice and Experience, vol. 29, pp. 1149-1171, 1999.

[20] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," in Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, 2003.

[21] D. K. Kim, J. S. Sim, H. Park and K. Park, "Linear-time construction of suffix arrays," in Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003, Morelia, Michoacán, Mexico, June 25-27, 2003. Proceedings, pp. 1017-1017.

[22] J. Kärkkäinen and P. Sanders, "Linear work suffix array construction," Journal of the ACM, pp. 918-936, 2006.

[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008, pp. 73-82.

[24] C. Trapnell and S. L. Salzberg, "How to map billions of short reads onto genomes," Nat. Biotechnol., vol. 27, pp. 455-457, 2009.

[25] M. Pop, "Genome assembly reborn: recent computational challenges," Briefings in Bioinformatics, vol. 10, pp. 354, 2009.

[26] S. Popov, J. Gunther, H. P. Seidel and P. Slusallek, "Stackless kd-tree traversal for high performance GPU ray tracing," in Computer Graphics Forum, 2007, pp. 415-424.

[27] J. C. Venter, "Multiple personal genomes await," Nature, vol. 464, pp. 676-677, 2010.

[28] J. Kaiser, "DNA sequencing: A plan to capture human diversity in 1000 Genomes," Science, vol. 319, pp. 395, 2008.