Parallel Gene Upstream Comparison via Multi-Level Hash Tables on GPU

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Andrew Todd

Dr. Michela Becchi, Thesis Supervisor

December 2016

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

Parallel Gene Upstream Comparison via
Multi-Level Hash Tables on GPU

presented by Andrew John Todd,

a candidate for the degree of master of science,

and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Michela Becchi

Professor Guilherme DeSouza

Professor William Harrison

To my wife.

ACKNOWLEDGEMENTS

This work has been supported by NSF award CCF-1421765 and CNS 1429294 and by equipment donations from NVidia Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The region of DNA immediately in front of a gene body (also called upstream region) contains short (8-20 base) sequence motifs that help to control when that gene is turned on and off. Unfortunately, these motifs are generally unknown and commonly degenerate.

In this work, a motif-finding framework is proposed that, given a set of gene upstream regions, performs all-to-all pairwise comparison and identifies all motifs of length *k* (k-mers) that are common to any pair of upstream regions or differ in at most *d* characters. This framework stores the k-mers found in each gene in a multi-level hash table. The hash table design optimizes hash table comparison (rather than hash table insertion or lookup), is highly parallelizable and easily maps onto GPU. Four GPU kernels are proposed for pairwise hash table comparison, each leveraging a distinct parallelization approach. A study of different factors that affect the performance of the implementations is included (the hash function, the number of buckets and the settings of additional implementation-specific parameters). Experimental results performed using an average-size yeast genome show that our fastest GPU kernel outperforms an 8-thread, cache-efficient CPU implementation by a factor of ~52x.

1. INTRODUCTION

A number of problems in computational biology involve the identification of short-to-moderate length strings that are identical or nearly identical between two or more sequences. These problems range from sequence database search [1], to genome assembly [2], to evolutionary genomics [3]. Hence, there is a significant need to evaluate the performance of computational tools for finding such near-exact matches on new hardware platforms. As a model application here, we consider the problem of understanding the short (8-20 base pair) sequences that govern how genes in a genome are turned on and off [4]. Genes are regulated by the binding of special *transcription factor* proteins to these specific short sequences of DNA, which we will refer to as "*motifs*" [5, 6]. These motifs are generally, but not invariably, located "in front" (upstream) of the gene they control.

Unfortunately, while target motifs for some transcription factors are known [7], most are not. Moreover, not all of the positions in the motif are critical for recognition, meaning that regulating motifs for the same transcription factor can differ somewhat from gene to gene [5, 6], making them difficult to locate using a single genome [8]. Instead, researchers have used a comparative approach to find regions of DNA that appear to regulate the same gene in several related organisms [4, 8, 9]. However, that approach suffers from two difficulties. First, even if a gene is regulated by the same transcription factor in two organisms, the specific motifs bound by that factor may move in the gene over evolutionary time. For instance, in humans and mice, the majority of binding sites have moved even when the same transcription factor is used [10, 11]. Even more problematic, the regulatory motif bound by a transcription factor may change over time [12], making the comparison of genomes challenging. However, with the advent of new sequencing technologies [13], the number of genomes available for such comparative genomic approaches is growing very rapidly. Already, there are dozens of genomes of closely-related fruit flies [14], yeasts [15] and mammals [16] available. Hence, developing comparative methods to overcome these two problems could very valuable.

To this end, we propose a comparative approach aimed to identify clusters of co-regulated genes based on the presence of *similar* motifs in the upstream regions of these genes, *without imposing constraints on the specific location* of these motifs within these upstream regions. Specifically, we propose a two- phase software pipeline. In the first phase, we compare all pairs of genes in each genome and identify the number of *similar* motifs of length *k* (*aka*, k-mers) between each pair of genes. In particular, we identify two k-mers as similar if they differ only in a limited number of positions (that is, if they are within a specific Hamming distance). We use the information collected in this first phase to build a similarity graph whose nodes represent genes and whose edge weights represent similarity scores between pair of genes. In the second phase, we use graph-based clustering techniques to identify groups of genes that share similar k-mers and are highly likely to be co- regulated by the same transcription factors.

We note that for the second phase of our pipeline, which is less computationally challenging, we can leverage existing graph clustering methods for CPU and GPU [17, 18]. Hence, in this paper we focus on the motif finding problem in the first phase of the pipeline. For each gene, our proposed solution stores the k-mers found in its upstream region in a set of hash tables (one for each Hamming distance considered). It then performs an all-to-all comparison of these hash tables to find and count common k-mers (that is, subsequences of length k). The execution time of this algorithm is dominated by the all-to- all hash table comparison, and we provide several GPU implementations to accelerate this operation. In this process, we study how hashing and work division among hash table buckets affects the performance of our parallel implementation.

Efficient hash table implementations for GPU have been proposed in literature and are publicly available [19, 20]. However, these solutions focus on optimizing the insertion and lookup operations on a single large and dynamic hash table, while we focus on optimizing the comparison of many small and static hash tables of variable size (which is dependent on both the k-mer variety and the length of the upstream regions of the genomes considered).

The proposed contributions are the following:

1.) A hash table-based, motif-finding framework that, given a set of gene upstream regions, performs their all-to-all pairwise comparisons and identifies all k-mer that are common to any pair of upstream regions or differ in at most d characters. The occurrence frequency counters produced by this framework are used to build a gene similarity graph for further analysis.

2.) A highly parallelizable multi-level hash table design that aims at accelerating hash table comparison (rather than hash table insertion or lookup), is amenable for GPU implementation, and encodes k-mers with different Hamming distances as 64- or 32-bit integers.

3.) Four GPU implementations of the all-to-all hash table comparison phase. The GPU kernels leverage distinct parallelization approaches and differ in their computation and memory access patterns. Therefore, the selection of the hash function, the bucket size and other kernel-specific parameters affect framework performance.

2. DESIGN OF PROPOSED MOTIF-FINDING FRAMEWORK

a. High-Level Algorithm Design

Recall that the considered motif finding problem can be formalized as follows. Given n sequences of lengths {l1, …, ln}, each representing the upstream region of a gene, for each sequence pair we want to find the common subsequences of length k (k-mers or motifs) with Hamming distance equal to or less than d. In other words, for each pair of sequences we want to find all k-mers that differ in at most d characters (where each character represents a base). Biologically relevant k-mer sizes vary from 8 to 20 for yeast. The yeast genome used as an exemplar in this study is an average-size genome for this group and has 4,592 extractable annotated genes. For each gene we consider both forward and reverse-strands of the upstream region, leading to 9,184 upstream regions (*n*). These upstream regions are constant at 500 bases long (*l*). Since we aim to find k-mers with high similarity, we consider small Hamming distances (≤ 2). While our approach is general and suited to a broad range of parameter settings, we present our results here in regards to the yeast genome in question relying on the scalability of our methods to make our work relevant for any desired sequence.

Recall that the output of the motif finding phase is a similarity graph whose nodes represent genes, and whose edge weights represent similarity scores between pair of genes. In turn, these similarity scores are weighted sums of the number of occurrences of k-mers that are common to each pair of genes, where identical k-mers shared between two genes are weighted highly and more dissimilar k-mers (i.e., k-mers with greater Hamming distance) are weighted less. Hence, we need a fast method to identify and count all common k-mers between each pair of genes.

A naïve solution to this problem could use an array of counters for each gene upstream. For the problem in consideration, with the four bases {A, C, G, T} there would be from 4^8 to 4^{20} possible k-mers. If arrays' slots are enumerated to contain 16-bit counters for each possible k-mer, this would require from 128 KB to 2TB per gene, resulting in a total of 3.36 GB to 55,104 TB for the considered genome and Hamming distance ≤2. Such requirements are space prohibitive for either NVidia GPUs (DRAM ≤12 GB) or CPU nodes. However, particularly for larger values of k, only a subset of all possible k-mers will be present in each genome. Further, the number of k-mers found upstream of a gene is limited by the length of that region. With small upstream regions, a radix sort based method might be practical, but not sustainable for growing sequences. Therefore, using a pointer-based chaining hash table allows for most scalable memory usage, particularly after the pointer-based structure is converted to an array-based representation, as shown in Figure 1.

Our implementation uses, for each gene, d Meta tables, each containing the k-mers found in that gene's upstream region and their occurrence frequency counters. The motif finding algorithm comprises two phases that can be pipelined. In the hash table construction phase, the input sequences (i.e., gene upstream regions) are streamed and the corresponding tables are created. Note that, for a given exact-match k-mer, there are $\left(\prod_{i=0}^{d-1}(k-i)\right)/d!$ partial match k-mers at Hamming distance d. For example, for d=1, we will have k distance-1 k-mers obtained by replacing a distinct character of the k-mer with a wildcard. Accordingly, for each k-mer found in the input, we will have one k-mer insertion or counter update in the exact-match table, and $\left(\prod_{i=0}^{d-1}(k-i)\right)/d!$ insertions or counter updates in each Hamming

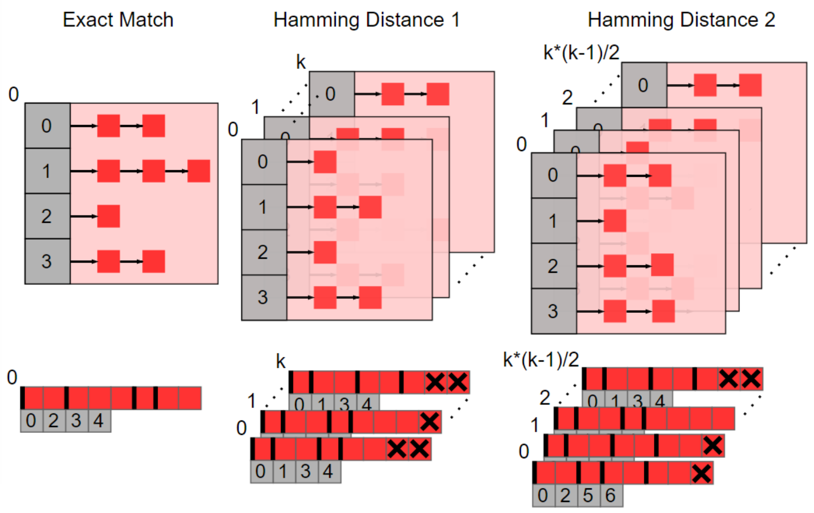
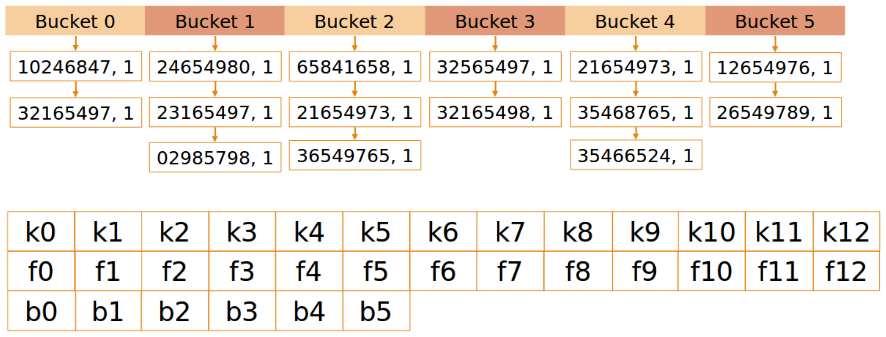


Fig. 2. Hamming distance meta-tables and their respective sub-tables before and after array based conversion.



k0	k1	k2	k3	k4	k5	k6	k7	k8	k9	k10	k11	k12
f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
b0	b1	b2	b3	b4	b5							

Fig. 1. (Top) Pointer-based chaining hash table containing a set of k-mers. The unsigned tuple of values in the buckets indicate the 64-bit compressed representation of each k-mer, and an occurrence frequency counter. (Bottom) Compressed array-based representation of the same hash table, where k_i represent the kmers, f_i their occurrence frequency counters, and b_i the offsets of the buckets within the arrays of k-mers and counters.

distance-d table. In the hash table comparison phase, the all-to-all pairwise comparison of the hash tables will allow finding the number of k-mers shared by each pair of gene with a given Hamming distance. Figure 2 provides a high level representation of the data structures used in our implementation. Below, we provide details on our design.

b. *Multi-Level Hashing*

In principle, if wildcards are encoded as special characters, all exact-match and distance-*d* k-mers can be stored into a single table. However, this solution would have two drawbacks. First, the table would become dense and insertion time would grow with more time required to traverse each bucket either looking for an entry to increment or adding a new unique entry. Second, recall that the similarity scores generated at the end of the motif-finding phase must give lower weight to more dissimilar k-mer matches. To this end, we need to keep distinct counters for matches with different Hamming distance. As a result, in the comparison phase, a unified hash table would make it necessary to process k-mers found in the same bucket but containing a different number of wildcards differently. This would add control flow operations to the comparison code, possibly leading to warp divergence and inefficiencies in a GPU implementation. In order to avoid these inefficiencies and simplify the GPU code, we adopt a multi-level hash hierarchy, with each class of match allowed its own meta-table (Figure 2). Each meta-table represents a Hamming distance class. For $d > 0$, the distance-*d* meta-table consists of $(\prod_{i=0}^{d-1}(k-i))/d!)$ sub-tables, each corresponding to a wildcard configuration. For example, sub-table 0 for the distance-1 meta-table will allow entries for k-mers with a wildcard in first position; similarly, sub-table 2 will store k-mers with a wildcard in second position. For the distance-2 meta-table, two wildcards are possible within each enumerated sub-table. Note that insertions in different sub-tables can be performed in parallel, allowing for a straightforward parallelization of the hash table construction phase. The parallel nature of using many sub-tables also makes the GPU version feasible, though with nuances described in III.

c. K-mer Encoding

In the most straightforward encoding, a k-mer is represented as an ordinary string (array of chars). This encoding allows arbitrarily long k-mers and supports the entire set of DNA ambiguity characters (where characters like R or Y represent the purine or pyrimidine base-classes respectively, or N indicates a completely indeterminate base). However, performing motif finding on sequences with such undetermined sequences could generate biologically spurious results. Accordingly, given our interest in k values between 8 and 20, we decided to adopt more space and time efficient representations by ignoring parts of the upstream sequences with ambiguous bases and using the upper bound of k=20 to structure our motif representation.

Specifically, we implemented a 64- and a 32-bit k-mer encoding. The 64-bit scheme encodes each symbol using 3 bits, retaining some flexibility in encoding special characters and allowing a maximum k-mer size of 20. The 32-bit representation encodes each symbol using 2 bits, limiting the maximum k-mer size to 15 and disallowing ambiguous bases. For the above encodings, wildcards within non-exact-match k-mers are represented as one of the four bases (namely A). The use of the multi-level hashing scheme just described allows distinguishing wildcards from regular bases implicitly. In these two encoding schemes, four and two reserved bits respectively, function as algorithmic bookkeeping counters used for programming convenience. We experimentally observed that, on CPU, using a 64-bit encoding allows a speedup in the order of 8x over a char-based representation, while using 32-bit encoding does not lead to significant performance improvements over 64-bit encoding.

d. Bucket Representation

For hash table construction on CPU we use a pointer-based bucket representation (top of Figure 1). In addition, we insert k-mers into buckets in a sorted fashion, so to reduce comparison times later. Importantly, it is well known that pointer chasing on GPU may lead to highly inefficient global memory accesses. To this end, before transferring the hash tables onto GPU for comparison, we convert all sub-tables into array form (bottom of Figure 1 and 2). Fixed-length arrays make it possible to align, cache, and calculate memory offsets for any upstream region, which is highly desirable in GPU kernels. Furthermore, in the interest of a fair analysis between CPU and GPU comparison, we implement a CPU comparison version that utilizes the same fix-length array structure. For analogous alignment and caching benefits [21], this allows a reasonably optimized CPU performance to measure against.

e. Algorithmic Analysis – Construction vs. Comparison

The number of k-mers that can be found in a sequence of length l is $(l\text{-}k\text{+}1)$, so the corresponding hash table creation involves $(l\text{-}k\text{+}1)$ k-mer insertions/updates. Since the buckets are sorted, bucket insertions and updates happen in linear time with respect to the size of the bucket. In the worst case (a single bucket), hash table construction for n input sequences (gene upstream regions) can be done in $O(n(l-k+1)^2)$ time. The number of pairwise comparisons required on n hash tables is $n(n-1)/2$. Since buckets are sorted, comparing two buckets can be done in linear time. Hence, the all-to-all pairwise comparison of n single-bucket hash tables can be performed in $O(n(n-1)(l-k+1))$ time. For $n>>l$, as in the case of the yeast genome considered, the hash table comparison phase dominates the execution time. This prediction is confirmed by experiments performed on upstream regions from this genome using an 8-threaded implementation on an 8-core CPU. When increasing n from

1,024 to 2,048 and then to 4,096, the table construction time increases linearly from ~2.5s to 5s to 10s, while the table comparison time increases quadratically from ~11s to 42s to 155s. Hence, in this work we focus on the GPU acceleration of the hash table comparison phase, allowing pipelining of CPU table construction and GPU table comparison.

f. Hashing Methods

For hash table construction, we tested the following hash functions (their bit grouping schemes are shown at the end of this section in Figure 3).

1) DJB2: We chose the djb2 hashing algorithm for its simplicity. The little understood technique to this algorithm is simply using primes 33 and 5,381 as follows:

```
hash(unsigned int kmer){
    hash = 5381;
    for all “ 3 (or 2) bit groupings”{
        hash = hash*33+“current bit grouping”
    }
    bucket_in_table = hash % buckets_per_table
    return bucket_in_table;
}
```

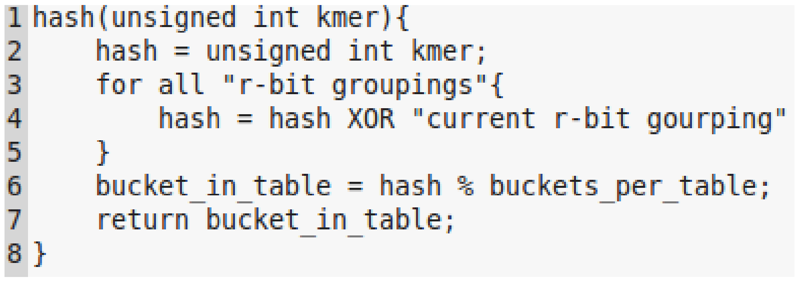
In this case, *bit groupings* are the bit-encoded nucleotide bases A, C, G, and T. Whether encoded by a 2-bit or 3-bit scheme, the algorithm works the same, with a sequence of bit groupings packed into a 32-bit or 64-bit unsigned integer.

2) 2-U MT: Similar to Alcantara in his CUDPP hashing contribution [19, 20], we used a 2-universal hash function of the form:

```
hash(unsigned int kmer){
    hash = (a XOR k + b)mod P mod B;
    bucket_in_table = hash % buckets_per_table;
    return bucket_in_table;
}
```

Above, *k* is the key, *a* and *b* generated by the Mersenne Twister PRNG, *p* is the largest 32/64-bit unsigned prime, and *B* is the number of buckets in a sub-table. In this case, the key is the entire 32/64-bit unsigned integer representing the k-mer sequence.

3) Custom Xor: Lastly, we drew from both djb2 and 2-U MT, disregarding pre-defined constants, grouping by bits across nucleotide boundaries, and opting to use XOR. The size of bit groupings, r, corresponds to the max number of buckets desired (4-bits for 16 buckets up to 8-bits for 256 buckets). This method operates as follows:



```
hash(unsigned int kmer){
    hash = unsigned int kmer;
    for all "r-bit groupings"{
        hash = hash XOR "current r-bit gourping"
    }
    bucket_in_table = hash % buckets_per_table;
    return bucket_in_table;
}
```

Above, *hash* starts initially with the input compressed k-mer. We found this method to give the best performance and predictability for our dataset, as demonstrated in Section VI.

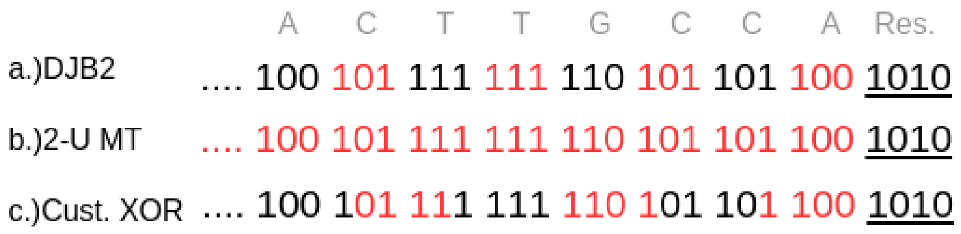

A C T T G C C A Res.
a.)DJB2 .... 100 101 111 111 110 101 101 100 1010
b.)2-U MT 100 101 111 111 110 101 101 100 1010
c.)Cust. XOR 100 101 111 111 110 101 101 100 1010

Fig. 3. Alternate red and black groups for (a) and (c) indicate bit groupings for hash function iterations. A single group of 60-bit wide operations occurs in (b). Spaces indicate indicate base boundaries.

3. GPU PARALLELIZATION OF HASH TABLE COMPARISON

In this section, we describe our GPU implementation of the hash table comparison phase, the most computationally intensive stage of our motif-finding pipeline. Since the hash function used for a single run of the pipeline is the same for all upstream tables, pairwise comparisons can be done on a bucket-to-bucket granularity, rather than requiring comparisons across buckets. Moreover, if buckets are filled in a sorted manner during construction, two buckets can be compared in linear time.

For a CPU implementation, splitting up work among threads at the upstream comparison task level provides significant (almost linear) speedup over the single-threaded version.

a. Background on GPU architecture and programming

Our GPU implementation is written using CUDA [22]. CUDA exposes the GPU architecture to the programmer and requires her to write code with greater awareness of the underlying hardware. First, NVidia GPUs consist of several SIMT processors, called Streaming Multiprocessors (SMs), each containing a set of in-order scalar cores. On the software side, CUDA structures the computation in a hierarchical fashion: it groups threads into thread-blocks, and maps each thread onto a core and each thread-block onto a SM. Therefore, to exploit the hierarchical hardware, the programmer must parallelize the work at different granularities and often prioritize one granularity over another. Second, due to the SIMT nature of the SMs, threads execute in groups called warps on 32-element SIMT units. In every clock cycle, threads belonging to the same warp must execute the same instruction. As a result, in the presence of control-flow operations, full core utilization requires warps to follow the same execution path. Otherwise, each alternate path will execute in serial, reducing execution efficiency by a factor of the number of diverging branch operations. Third, GPUs have an explicitly managed memory hierarchy. They are equipped with a relatively large off-chip, high-latency, read-write global memory (commonly called global memory); a smaller low-latency, read-only constant memory (which is off-chip but cached); and a limited on-chip, low-latency, read-write shared memory, which can be used either as a software or as a L1 hardware cache. The global memory can be accessed via 32-, 64- or 128-byte transactions and has a high access bandwidth (up to about 330 MB/s). Since contiguous memory accesses can be coalesced into single memory transactions when accessing global memory and can prevent bank conflicts when accessing shared memory, proper bandwidth utilization requires threads to access adjacent data rather than in the individual chunks that CPU multithreading prescribes. In addition, although some caching automatically occurs via a limited hardware L1, the more performance-vital caching often occurs in a software managed fast memory that possesses much larger capacity per streaming multiprocessor (SM).

b. GPU kernel design considerations

In order to design an efficient GPU implementation we need to consider all GPU characteristics mentioned above: the parallelization approach (i.e., work division and distribution among threads and thread-blocks), the potential for warp divergence and the memory access patterns. We discuss each in turn.

Parallelization approach: The computation considered exhibits parallelism at different granularities. At the coarse grain, given *n* upstream regions, we are presented with *n*(n-1)/2* pairwise comparisons that can be done in parallel. Each upstream region is associated *d* hash tables, resulting in $1 + \sum_{j=1}^{d}((\prod_{i=0}^{j-1}(k-i))/j!)$ sub-tables; pairs of sub-tables from different upstream regions can also be processed in parallel. Each sub-table, in turn, consists of *b* buckets. As mentioned above, pairwise comparisons can be done on a bucket-to-bucket granularity, and distinct pair of buckets can be processed in parallel. Finally, the bucket-level comparison can be parallelized by having groups of threads cooperatively compare different elements within the pair of buckets: this parallelization, however, is not work efficient and leads to extra comparisons that would be avoided in a serial implementation at this level.

Warp divergence: We note that, since different buckets may have different sizes and different number of common elements, the bucket-to-bucket comparison has a data-dependent and irregular computational pattern, which can lead to warp divergence. In addition, the number of buckets used may affect the efficiency of the code. Recall that, by limiting the number of collisions, a large number of buckets can make hash table creation faster. However, the number of buckets affects hash table comparison in two ways: on one hand increasing the number of buckets increases the number of bucket-to-bucket comparisons to be performed, on the other hand, it leads to smaller buckets, possibly reducing warp divergence during comparison. Thus, the optimal number of buckets in the hash table comparison phase depends on the parallelization approach adopted and may differ from the optimal number of buckets for the hash table creation phase. It is worth noting that, after creation, the number of buckets can be efficiently reduced by combining buckets so long as bucket merging combines entire buckets and avoids splitting them. Hence, using a large number of buckets during creation provides more flexibility in the comparison phase. Finally, as anticipated in Section II.B, we observe that our hierarchical table design allows us to treat buckets in a uniform way independent of the meta-table they belong to (the meta-table information affects only the specific counters to be updated with the result of the comparison). Besides leading to simpler code, this design decision eliminates one potential source of warp divergence.

Memory access patterns: We observe that the array-based hash table representation shown in the bottom of Figure 1 and 2 allows coalesced global memory accesses, since adjacent threads can access contiguous elements within each bucket. However, whether performed in serial by a single thread or in parallel by multiple threads, bucket-to-bucket comparison does not allow coalesced memory accesses. As a result, the use of shared memory is an important factor in performance. In all our GPU implementations, we first read bucket data into shared memory in a coalesced fashion as much as possible, and then we perform bucket-to-bucket comparison directly in shared memory. In addition, to save shared memory space and make it available for the bucket data, we force the bucket offsets (the third array in the bottom part of Figure 1) to be stored in L1 cache by using the *__ldg* intrinsic.

c. GPU kernel implementations

We consider two approaches to the all-to-all hash table comparison problem. The first approach seeks to minimize warp divergence and optimize memory access patterns at the cost of sacrificing parallelism. The second approach seeks to maximize parallelism at the expense of increased warp divergence. In the first, hierarchical approach (cooperative execution), each pairwise comparison of gene upstream regions is mapped to a thread-block, and the threads within the same thread-block perform cooperatively the bucket-to-bucket comparisons for all the sub-tables associated to the given pair of genes. In the second, flat approach, each pairwise comparison of gene upstream regions is mapped to a thread, and each thread performs the bucket-to-bucket comparisons for the considered pair of genes sequentially. For the cooperative execution approach, we provide three implementations (Sequential Bucket, Thread to Bucket, and Warp to Bucket kernels), which differ in the way they distribute the work within a thread-block. For the flat approach, we provide only one version (Thread to Pair kernel). Figure 4 illustrates the parallelization approach of the four kernels. Below, we provide more detail on their implementation and discuss their relative strengths and weaknesses.

Sequential Bucket: We recall that, in this approach, each pairwise comparison of upstream regions is assigned to a thread-block, and the threads within the same block perform each bucket-to-bucket comparison cooperatively. Each bucket-to-bucket comparisons within a sub-table are performed sequentially by the thread-block. Before performing the comparisons, the threads cooperatively load the entire upstream pair into shared memory. If two hash tables under consideration exceed the shared memory capacity, they can be loaded and processed in stages (not shown in performance section for brevity). The pairwise comparison between buckets Bi and Bj is parallelized as follows. Each thread of a warp is assigned one or more k-mers from Bi in a contiguous fashion and compares such k-mers with all k-mers in Bj sequentially. Because the bucket keys are sorted, a slightly different version of the kernel avoids some k-mer comparisons by using a linear time approach. In both cases, however, this kernel performs some extra k-mer comparisons which would not be required if the bucket-to-bucket comparison was executed entirely in serial. Warp divergence in this case is reduced since different threads tend to perform similar amounts of work.

Warp to Bucket: This version is similar to the Thread to Bucket kernel, except that it assigns different warps to different buckets. The threads within a warp will process each bucket cooperatively. In terms of work efficiency and warp divergence, this kernel is intermediate between the sequential-bucket and the Thread to Bucket versions.

Thread to Bucket: This version is similar to the Sequential Bucket kernel, except that it assigns different threads within a block to different buckets. If the buckets of the two hash

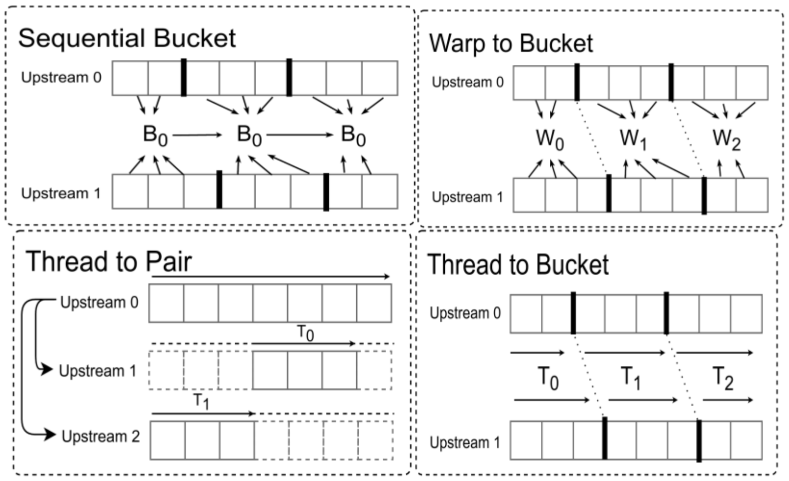


Fig. 4. Illustration of the four proposed kernel implementations. In the figure, *upstream i* represents the set of hash tables associated with gene *i*, and *Bj*, *Tj* and *Wj* represent block *j*, thread *j* and warp *j*, respectively. The bolded lines represent bucket boundaries (the bucket size may vary from bucket to bucket). The *Sequential Bucket*, *Thread to Bucket* and *Warp to Bucket* kernels map pairwise comparisons to thread-blocks, and let the threads within the block cooperatively process buckets in different fashions: either all threads in a block process a bucket cooperatively, or different threads process different buckets, or different warps process different buckets. *The Thread to Pair* kernel maps pairwise comparisons to threads, and threads within the same block share one of the upstream regions of the pair.

tables under consideration exceed the shared memory capacity, they can again be loaded and processed in stages. In this case, however, shared memory must hold a segment from each bucket (since buckets are processed in parallel by the threads). To allow coalescing of memory accesses, the size of the segments loaded into shared memory at each stage must be a multiple of 128 bytes. Since each bucket-to-bucket comparison is performed sequentially by a thread, this implementation performs only the k-mer comparisons that are strictly required. However, bucket imbalances may in this case lead to varying degrees of warp divergence. Hence, a fair hash function is vital.

Thread to Pair: As explained above, this kernel maps each pairwise comparison of upstream regions to a thread, and each thread performs the bucket-to-bucket comparisons for the considered pair of upstream regions sequentially. Each thread-block, then, works on multiple pairwise comparisons, all sharing the same first upstream region in the pair. For each sub-table, we merge all the buckets into a single bucket. Hence, for each sub-table, each thread will perform a maximum of 2(l-k+1) comparisons, where l is the length of the upstream regions. The computation is broken down into small configurable chunks of cacheable sub-arrays of k-mers. Each thread fetches and caches the next chunk into shared memory whenever it reaches the end of the current chunk. This method has the obvious drawback of being inefficient, because all threads that are doing comparisons have to wait for a single thread that reaches the end of the chunk. In addition, this method may suffer from significant warp divergence. On the other hand, this approach can perform a massive number of pairwise hash table comparisons (as many as the number of threads launched) in parallel.

4. RELATED WORK

a. *Automata-based approach*

The motif-finding problem considered can be also addressed through a finite automata-based approach. Specifically, given an upstream region U_i of length l, the main idea is to construct the finite automaton that accepts all the k-mers found in U_i within Hamming distance d [23]. The k-mers that are common to U_i and any other upstream region U_j (or that differ in at most d characters) can then be found by traversing the constructed automaton using input U_j. In general, a non-deterministic finite automaton (NFA) that accepts a string of length k with Hamming distance d consists of $(k+1)(d+1)-d(d+1)/2$ states. An upstream region of length l requires $(l-k+1)$ such NFAs. The all-to-all pairwise comparison of n upstream regions would require an NFA for all but one upstream region. For the problem size under consideration here (Section II.A), these requirements would lead to a total number of states ranging from about 108 million (for k=8) to 265 million (for k=20), and a total number of transitions ranging from about 299 million (k=8) to 768 million (k=20). Parallel implementations of NFA traversal are available for several platforms including FPGAs [24], GPUs [25], and Micron's Automata Processor (AP) [26, 27].

Logic-based FPGA designs [24] provide NFA traversal throughputs in the order of several Gigabit/s. These implementations are based on the one-hot encodings scheme, which encodes each NFA state in a flip-flop. Large Xilinx FPGAs have in the order of 100 thousand flip-flops. Therefore, for the problem size considered, we would need either to use one-to-three thousand FPGAs or to stage the computation on one or few FPGAs. The main problem of these logic-based implementations is that loading an NFA on FPGA requires running full synthesis and place-and-route, which can require minutes to hours. Therefore, these FPGA designs are impractical for the problem at hand.

To support the number of NFA states and transitions above, the GPU-based NFA processing engine proposed by Cascarano et al. [25] would require from 2.23 GB (for *k*=8) to 5.73 GB (for *k*=20) of global memory and from about 26 MB to 63 MB of shared memory (the latter to store the active state vectors). While the NFA data structure itself would fit in the 12 GB of global memory available on current NVidia GPUs (e.g., Titan X), the active state vectors would exceed the currently available shared memory (i.e., 48KB per block), requiring a kernel redesign or staging the computation into between 500 and 1,000 iterations. Using Micron AP's encoding, an NFA accepting a string of length *k* with Hamming distance *d* would require $(2d+1)k-d^2$ AP state transition elements (STEs) [27]. As a consequence, the problem in hand would require from about 163 million to about 424 million states (for *k*=8 and *k*=20, respectively). With an AP chip containing 49,152 STEs, the problem at hand would require from 3,315 to 8,628 AP chips (for *k*=8 and *k*=20, respectively), assuming that the interconnect does not limit the AP chip utilization. With 48 chips on an AP board, this would require staging the computation into between 70 and 170 iterations. GPU and AP implementations have been exist focusing on applications for which the NFA construction and preprocessing times are not an issue. However, this is not the case for the problem considered. For both platforms, the required NFAs must first be pre-computed on CPU and encoded in a format suitable for GPU or AP deployment and then offloaded to the device. To allow for efficient reconfiguration, the AP interconnect must be preconfigured to support the required NFA topology. In addition, Micron's AP has a significant overhead associated with processing the output (match) information. This overhead could significantly affect the performance.

b. Statistical motif finding approaches

Most of the existing algorithms in this focus-area find possible motifs by using statistical methods to locate overrepresented k-mers [28]. Two common approaches involve either word-based algorithms or heuristic algorithms. Word-based algorithms find overrepresented k-mers of fixed length for all available upstream regions by iterating through the text. This method is shown to be resource-intensive, but exhaustive. Moreover, effectiveness prevails only for short, conserved motifs. The second approach uses statistics to determine sites with high probability of significance. The method is less resource-intensive and applicable to long motifs, yet is not exhaustive.

c. Other hash table implementations for GPU

Partly following the model of Lefebvre and Hoppe [29], we construct hash tables on the CPU and access them on the GPU. In contrast, Alcantara et al [19] pioneered a hash table approach performing both hash table construction and access directly on the GPU. The cuckoo hashing implementation of Alcantara's influential paper has a production version in the open source CUDPP library (CUDA Data Parallel Primitives), however this code is optimized with hash tables of 10K+ entries in mind whereas we use many small tables with only 500 unique entries.

In our method, we chose to use a multi-level hash table scheme to allow convenient storage of separate classes of partial matches that are relevant biologically. In order to employ the cuckoo method above, we could have instead used a single-level hash table to represent an upstream where all exact and partial matches hash to unique keys. The number of entries of this table would then range from 18,500 to 105,500 for k-mer lengths of 8 and 20 respectively, well inside the range of optimization of the cuckoo method. However, the Cuckoo hashing of CUDPP would require query based access, rather than direct contiguous array access because of the key displacement protocol used in cuckoo hashing. This displacement dictates that hash collisions move keys/values to disparate memory locations to avoid further collisions. Such displacement would destroy caching performance and coalesced global memory access, disallowing our array-to-array comparison on GPU.

5. RESULTS

a. Hardware and Software Setup

We run all experiments on an 8-core Intel Xeon CPU E-5620 v3 @ 2.40GHz with 64GB of RAM. The machine is equipped with an NVidia Titan X GPU with 12GB of RAM. The GPU is of Maxwell generation, has 24 streaming processors, and each streaming processor has 128 cores and 96KB of shared memory. The GPU runs at 1,088 MHz. The software setup is 64-bit CentOS release 6.4 with NVidia CUDA version 7.0.

b. Dataset

We extract the dataset from the Yeast Gene Order Browser [15] E. Cymbalariae genome. This genome has 4592 extractable genes and is average in terms of the number of genes among the yeast genomes. For each gene we consider both forward and reverse-strands of the upstream region, leading to 9,184 upstream regions. In accordance with Harbison et al. [30], the extracted upstream regions are 500 bases long.

c. Hash Table Construction Performance

With any tool that uses hash tables, it is useful to consider the fairness of hash functions used in order to preserve desired space and time characteristics. Specifically, we examined how serial insertion time depends on hash function and bucket configuration. Not only is the fairness important inasmuch as it affects performance, but also the balanced bucket populations given by a fair hash function will help load balance the comparison task later offloaded to the GPU. Figure 5 reports the behavior of three approaches to hashing,

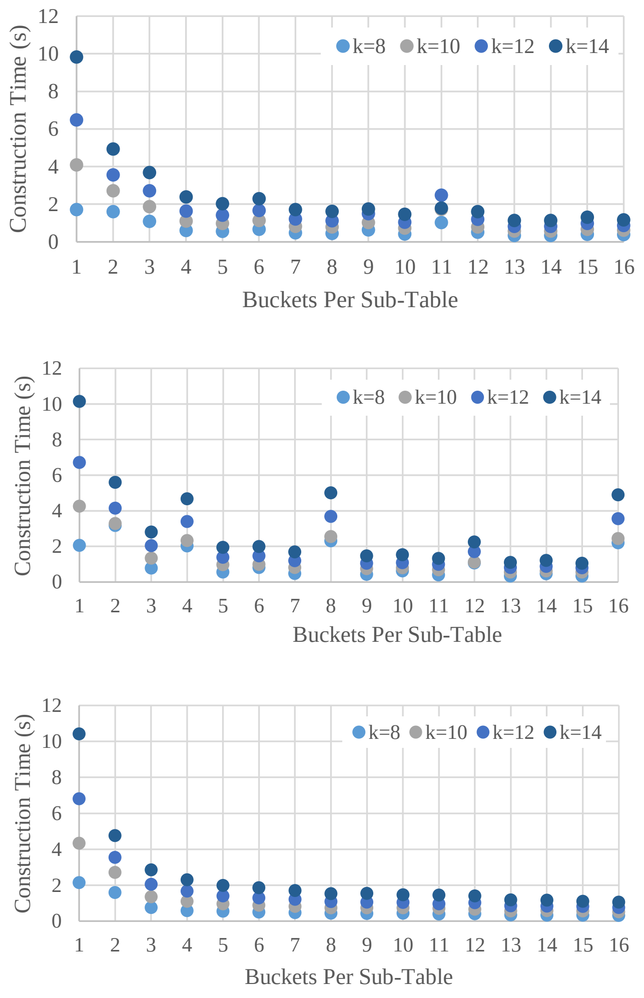


Fig. 5. Construction time for three hash functions across different bucket configurations and k-mer sizes. Spikes in construction time correspond to hash unfairness. (top) DJB2-based construction. (middle) 2U-MT-based construction. (bottom) Custom XOR-based construction.

highlighting the strength of our custom XOR method in terms of reliable construction performance for various bucket sizes (and by implication, fairness). Note in Figure 5 how irregular construction times result for DJB2 and 2U-MT for certain buckets per sub-table settings (BPS). DJB2 sees a decrease in performance for BPS = 11, and 2U-MT sees decreases for all base two BPS settings and BPS = 12. This phenomenon is due to the inability of the hash functions to provide fairness (i.e. balanced bucket populations) for those particular bucket counts. Additional data (not included for brevity) show high variance across bucket populations for those configurations where hash construction time worsened. In this data, the custom XOR hash function maintained comparatively low variance in all bucket configurations making it the most reliable choice for our pipeline. Again, bucket unfairness not only makes for decrease in hash table construction time, but more importantly causes load imbalance among processing elements of the GPU due to static work assignment of Blocks, Warps, and/or Threads to bucket to bucket comparison tasks. In other words, some elements must synchronize after completing smaller bucket-to-bucket comparisons, while others continue working.

d. GPU Hash Table Comparison Performance

Figures 6 and 7 report performance results for our hash table comparison kernels. The first two GPU implementations require similar kernel configuration styles, while the second two require specially tailored configurations based on the optimizations they employ. For the

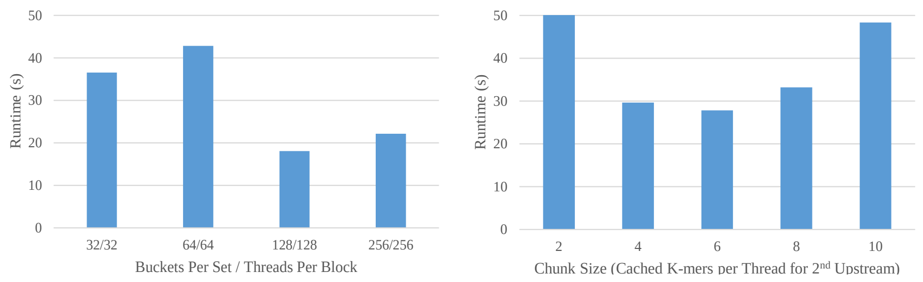


Fig. 6. Execution time of the *Thread to Bucket* kernel with both upstreams fully cached (to the left); and execution time of the *Thread to Pair* kernel with one upstream fully cached and the second cached up to Chunk Size for each thread, with block size of 128 (to the right). Both versions use 1024 blocks with k = 12.

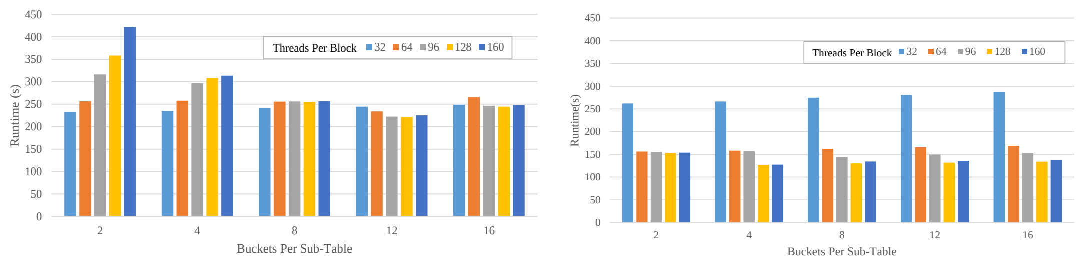


Fig. 7. Execution time of the *Sequential Bucket* kernel (left) and of the *Warp to Bucket* kernel (right) for different bucket settings and block sizes.

Sequential Bucket implementation (left side of Figure 6) we note for small BPS configurations that hardware underutilization increases with higher block size. This is because if all warps in a block must compare a single bucket pair on a streaming multiprocessor, some miss the opportunity for simultaneous comparison of other bucket pairs of the upstream while tied up in synchronization cycles. This results in less overall parallelism and memory bandwidth underutilization when accessing upstream hash tables in shared memory.

For the *Warp to Bucket* implementation (right side of Figure 6) we note that having a single warp assigned to a comparison with multiple buckets is equivalent to the case of mapping a single block with block size 32 to a bucket, and accordingly performs on par with the version above. The remaining configurations where multiple warps map to their own respective buckets simultaneously perform better overall as expected, with the best performances obtained with many buckets and many warps to work on those buckets. Shared memory utilization is more efficient because more warps have opportunity to access shared memory.

For the *Thread to Bucket* implementation (left side of Figure 7), each thread within a block is assigned its own bucket. With warps scheduled as units rather than individual threads as units, to avoid warp underutilization the block size was set to a multiple of 32. Correspondingly, the buckets per sub-table were also set to be equal to the number of threads so that each thread could process its own independent bucket. We note an optimum at 128/128, a consequence of the Titan X GPU having 128 cores per streaming multiprocessor. Though this method ultimately reduces both the effective shared memory bandwidth usage at the warp level and the warp efficiency, more total warps execute simultaneously.

Therefore, the larger number of resident warps on a streaming multiprocessor actually lead to higher effective bandwidth usage overall.

For the *Thread to Pair* implementation (right side of Figure 7), each thread is assigned a pair of upstream regions. Limiting factors of this version include lower global memory read efficiency and block-level synchronizations, both of which are required to facilitate its unique k-mer caching scheme. Recall the Thread to Pair implementation caches one whole sequence - the “fixed sequence” - in shared memory, and then fits chunks of every other sequence. Then, the loop iteratively compares those chunks with the said fixed sequence. The platform has peak theoretical occupancy at 6KB of shared memory per block for this implementation. In the case of 32-bit k-mers, the fixed sequence occupies 512 k-mers*32 bits per k-mer or 2 KB, hence each thread will have cache space of (4KB/128/32) bits per k-mer or 8 k-mers per chunk. Similarly, in the case of 64-bit k-mers, the calculation results in exactly 2 k-mers cached per chunk. The problem with short, cached chunks is that all threads in the same block have to stall more frequently to allow threads to cache more chunks in shared memory when needed. This stall leads to performance degradation. Therefore, even with 100% theoretical occupancy, the actual occupancy is closer to 30%. We tested both the 32 and 64 bit implementations with different chunk settings from 2 to 12, and the performance peaked at 6 k-mers per chunk for 64-bit (right of Figure 7) and 8 k-mers per chunk for 32-bit. We note that for 64-bit implementation, this is neither the longest chunk setting nor the 100% theoretical occupancy point. We conclude that 32-bit implementation is favored for analysis pipelines that require looking at sequences with longer k-mers. 64-bit k-mers work well, albeit less performant than 32-bit. Regardless of 32 or 64-bit usage, the kernel configuration of 128 threads per block with 1024 blocks gives the best performance for the kernel.

e. Final Speedup of All Comparison Phase Versions

In Figure 8, we demonstrate the level of performance as compared to an 8-thread CPU comparison version. Both the Thread to Pair and Thread to Bucket versions abandon the prioritization of warp-efficiency for focus on task-parallelism. This is an improvement over the first two version that work cooperatively within warps to limit warp divergence. Moreover, as opposed to Thread to Pair, the Thread to Bucket version retains cooperation among warps as well as two fully cached upstreams. This version is able to focus on the memory in the cache and get the most out of it at the block level before moving on to a different comparison requiring different sequences to be loaded into shared memory.

It is important to note that for larger sequences requiring large cache spaces, GPU hardware with smaller shared memory configuration would not perform as well. For example, pre-Maxwell compute capabilities like Kepler and Fermi commonly have 48kb of shared memory rather than 96kb, and additionally require double the clock cycles for each shared memory transaction. Either acceptance of much lower achieved occupancy or finer grain chunking of shared memory is required for these generations of hardware.

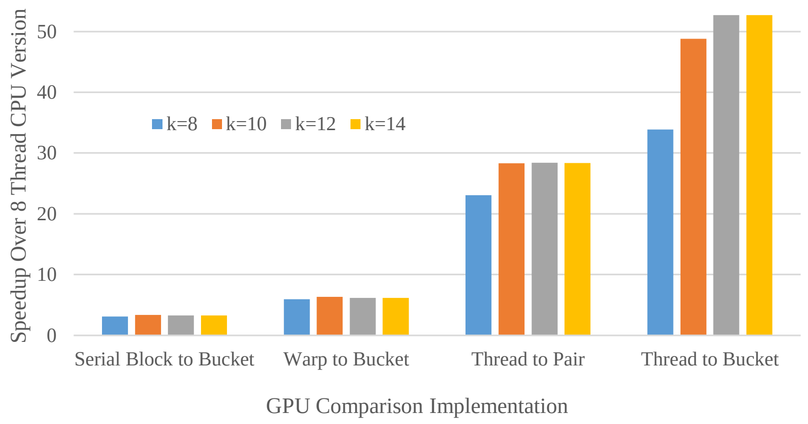


Fig. 8. Speedup over 8-thread CPU implementation for different k and GPU implementations. Upstream Comparison is done such that both CPU baseline and optimized GPU versions work on the same array-based data-structure.

f. Performance Scaling for Larger Upstream Sequences

To be generally applicable, the proposed comparison framework should maintain higher performance than the CPU version even when larger upstream sequences characterize the given genome. In Figure 9, the highest amount of speedup is seen in the upstream size of 500, with moderate dips in speedup for upstream size of 1000 and 1500. This can be attributed to increase in register count for the comparison kernels to allow cache chunking that caches portions of each upstream in succession for comparison. When the number of k-mers in a hash table required gets too high, they can't be cached and compared in a single run, but rather groups of buckets must be cached and compared in succession to keep shared memory usage low (and multiprocessor occupancy high). The results show that a decent amount of speedup over CPU comparison is attainable even for larger sequences, which is important because entire genes or genomes are often the subject of comparison rather than the much smaller upstream regions of E. Cymbalariae featured in this work.

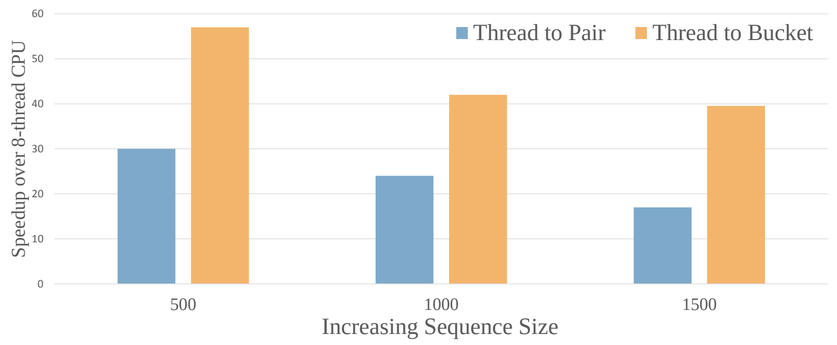


Fig. 9. Speedup over 8-thread CPU implementation for different k and GPU implementations. Upstream Comparison is done such that both CPU baseline and optimized GPU versions work on the same array-based data-structure. Though not biologically useful, upstream sizes of 1000 and 1500 of E. Cymbalariae were used in this test to allow for scaling analysis.

6. Conclusion and Future Work

In conclusion, we have proposed a motif-finding framework that, given a set of gene upstream regions, performs their all-to-all pairwise comparison and identifies all the motifs of length k (k-mers) that are common to any pair of upstream regions or differ in at most d characters. Our framework stores the k-mers found in each gene in a multi-level hash table. Our hash table design aims to optimize hash table comparison (rather than hash table insertion or lookup), is highly parallelizable and can be easily mapped onto GPU. We have proposed four GPU implementations of pairwise hash table comparison, each leveraging a distinct parallelization approach. We have performed an extensive experimental evaluation on an average-size yeast genome. Our results have shown that, for the considered genome, GPU implementations designed to optimize task parallelism perform better than kernels designed to limit warp divergence through synchronizations, and outperform an 8-thread CPU implementation by a factor of ~23 to ~52x.

In the future, we plan to use our framework on different genomes and perform a comparative intra- and inter-genome analysis with the goal of identifying genes that are regulated by the same transcription factors and thus share biological functions. From the computational standpoint, we expect that, on genes with longer upstream regions leading to larger hash tables, the Thread to Bucket version will be indispensable when coupled with shared memory chunking of both upstreams.

REFERENCES

[1] S. F. Altschul, *et al.*, "Gapped Blast and Psi-Blast : A new-generation of protein database search programs," *Nucleic Acids Research,* vol. 25, pp. 3389-3402, 1997.

[2] M. Pop, "Genome assembly reborn: recent computational challenges," *Brief Bioinform,* vol. 10, pp. 354-66, Jul 2009.

[3] J. Reneker, *et al.*, "Long identical multispecies elements in plant and animal genomes," *Proc Natl Acad Sci U S A,* vol. 109, pp. E1183-91, May 8 2012.

[4] D. Thompson, *et al.*, "Comparative analysis of gene regulatory networks: from network reconstruction to evolution," *Annual review of cell and developmental biology,* vol. 31, pp. 399-428, 2015.

[5] P. J. Farnham, "Insights from genomic profiling of transcription factors," *Nat Rev Genet,* vol. 10, pp. 605-16, Sep 2009.

[6] A. C. Meireles-Filho and A. Stark, "Comparative genomics of gene regulation-conservation and divergence of cis-regulatory information," *Curr Opin Genet Dev,* vol. 19, pp. 565-70, Dec 2009.

[7] V. Matys, *et al.*, "TRANSFAC: transcriptional regulation, from patterns to profiles," *Nucleic Acids Research,* vol. 31, pp. 374-8, Jan 1 2003.

[8] R. C. Hardison and J. Taylor, "Genomic approaches towards finding cis-regulatory modules in animals," *Nature Reviews Genetics,* vol. 13, pp. 469-483, 2012.

[9] M. Kellis, *et al.*, "Sequencing and comparison of yeast species to identify genes and regulatory elements," *Nature,* vol. 423, pp. 241-254, 2003.

[10] D. Schmidt, *et al.*, "Five-vertebrate ChIP-seq reveals the evolutionary dynamics of transcription factor binding," *Science,* vol. 328, pp. 1036-40, May 21 2010.

[11] D. T. Odom, *et al.*, "Tissue-specific transcriptional regulation has diverged significantly between human and mouse," *Nat Genet,* vol. 39, pp. 730-2, Jun 2007.

[12] G. P. Wagner and V. J. Lynch, "The gene regulatory logic of transcription factor evolution," *Trends Ecol Evol,* vol. 23, pp. 377-85, Jul 2008.

[13] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nature Biotechnology,* vol. 26, pp. 1135-45, Oct 2008.

[14] A. G. Clark, *et al.*, "Evolution of genes and genomes on the Drosophila phylogeny," *Nature,* vol. 450, pp. 203-218, 2007.

[15] K. P. Byrne and K. H. Wolfe, "The Yeast Gene Order Browser: Combining curated homology and syntenic context reveals gene fate in polyploid species.," *Genome Research,* vol. 15, pp. 1456-1461, 2005.

[16] P. Flicek, *et al.*, "Ensembl 2013," *Nucleic Acids Research,* vol. 41, pp. D48-55, Jan 2013.

[17] D. A. Bader and K. Madduri, "SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE Internat Symp on*, 2008, pp. 1-12.

[18] O. Kalentev, *et al.*, "Connected component labeling on a 2D grid using CUDA," *Journ of Parallel and Distributed Computing,* vol. 71, pp. 615-620, 2011.

[19] D. A. Alcantara, *et al.*, "Real-time parallel hashing on the GPU," presented at the ACM SIGGRAPH Asia 2009 papers, Yokohama, Japan, 2009.

[20] D. A. Alcantara, *et al.*, "Building an efficient hash table on the GPU," *GPU Comp. Gems,* vol. 2, pp. 39-53, 2011.

[21] N. Satish, *et al.*, "Can traditional programming bridge the Ninja performance gap for parallel computing applications?," pp. 440-451.

[22] J. Nickolls, *et al.*, "Scalable Parallel Programming with CUDA," *Queue,* vol. 6, pp. 40-53, 2008.

[23] K. U. Schulz and S. Mihov, "Fast string correction with Levenshtein automata," *Internat. Journ on Document Analysis and Recognition,* vol. 5, pp. 67-85, 2002.

[24] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *Proc. of the 4th ACM/IEEE Symp. on Architectures for Networking and Communications Systems*, 2008, pp. 50-59.

[25] N. Cascarano, *et al.*, "iNFAnt: NFA pattern matching on GPGPU devices," *ACM SIGCOMM Computer Communication Review,* vol. 40, pp. 20-26, 2010.

[26] P. Dlugosch, *et al.*, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 25, pp. 3088-3098, 2014.

[27] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th Internat*, 2014, pp. 415-424.

[28] M. K. Das and H.-K. Dai, "A survey of DNA motif finding algorithms," *BMC Bioinformatics*, vol. 8, pp. 1-13, 2007.

[29] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," presented at the ACM SIGGRAPH 2006 Papers, Boston, Massachusetts, 2006.

[30] C. T. Harbison, *et al.*, "Transcriptional regulatory code of a eukaryotic genome," *Nature*, vol. 431, pp. 99-104, Sep 2 2004.