HOMOLOGY SEQUENCE ANALYSIS USING GPU ACCELERATION

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

HUAN TRUONG

Dr. Gavin Conant, Thesis Supervisor

JULY 2018

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

HOMOLOGY SEQUENCE ANALYSIS
USING GPU ACCELERATION

presented by Huan Truong,
a candidate for the degree of Doctor of Philosophy and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Gavin Conant

Dr. J. Chris Pires

Dr. William Lamberson

Dr. Jianlin Cheng

For my parents, the humble educators.

ACKNOWLEDGMENTS

I would like to use this opportunity to express my gratitude to people who have supported me throughout my research.

My research career would not have been possible without my parents, who have believed in me in every step I took since college.

I would like to give my deepest gratitude to my advisor, Dr. Gavin Conant. Dr. Conant has not only been a great scientist with his insightful ideas and constructive criticisms, but also a supportive, patient, and gentle mentor. I would like to thank the current lab members Yue Hao and Rocky Patil for their support inside and outside the lab.

I would like to thank members of the committee, Dr. William Lamberson and Dr. Jianlin 'Jack' Cheng for many insightful inputs. I would like to personally thank Dr. Chris Pires and Dr. Michela Becchi for them always considering me being an "adopted" lab member. I was able to learn a substantial amount of knowledge from their lab members as well. Notably, I had the first tastes of late-night bench work from Sarah Unruh and Dustin Mayfield-Jones from Pires Lab; I have seen many algorithms and snippets of code from Dr. Daniel Li and Dr. Kittisak Sajjapongse (co-author in chapter 2), Andrew Todd (co-author in chapter 3), and Michael Butler from Becchi lab.

I was able to learn by looking up to former Conant lab members, Dr. Corey Hudson and Sara Wolff. During the year we shared the responsibility to be Teaching Assistants, Dr. Hudson has set an example of being an educator who makes hard concepts easy. There are many more graduate students in other labs who also helped me in many steps along the way, notably, Dr. Isaiah Taylor from Walker Lab and Samuel 'Sam' McInturf from Mendoza Lab.

I would like to thank the University of Missouri Informatics Institute administra-

tion staff, especially Dr. Chi-Ren Shyu, Robert Sanders, and Tracy Pickens, who have worked tirelessly over the years to support all students in the department. I would like to thank the Division of Animal Sciences, University of Missouri for hosting me for the five years of my study in Columbia, and the Bioinformatics Research Center, North Carolina State University for hosting me in the last year in Raleigh.

I thank NVIDIA Corporation for providing the critical components of my research depend on. This thesis would not be possible without their compilers, tools, and hardware.

Personally, I would like to thank Dustin Mayfield-Jones for being a supportive friend during and after his time at the University of Missouri. I thank Anh 'Gau' Pham for turning my life around when I was a college student in Vietnam. I thank Quyen Nguyen, Dr. Donald Bindner, Amy Nunan and Lathe Nunan for supporting me through the years, and Ly Vuong for the mental support I needed towards the completion of this thesis.

In the limited space of this acknowledgment, I am not able to list all the names of people I owe my gratitude to. I thank everyone who has worked with me or helped me during my years as a graduate student. I thank the world to allow me to stand on its shoulder, and I hope to contribute back to it one day in the future.

Lastly, the layout of this thesis is modified from an open-source template written by John Hammond and all mistakes in this thesis are of my own.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

A number of problems in bioinformatics, systems biology and computational biology field require abstracting physical entities to mathematical or computational models. In such studies, the computational paradigms often involve algorithms that can be solved by the Central Processing Unit (CPU). Historically, those algorithms benefit from the advancements of computing power in the serial processing capabilities of individual CPU cores. However, the growth has slowed down over recent years, as scaling out CPU has been shown to be both cost-prohibitive and insecure. To overcome this problem, parallel computing approaches that employ the Graphics Processing Unit (GPU) have gained attention as complementing or replacing traditional CPU approaches. The premise of this research is to investigate the applicability of various parallel computing platforms to several problems in the detection and analysis of homology in biological sequence. I hypothesize that by exploiting the sheer amount of computation power and sequencing data, it is possible to deduce information from raw sequences without supplying the underlying prior knowledge to come up with an answer. I have developed such tools to perform analysis at scales that are traditionally unattainable with general-purpose CPU platforms.

I have developed a method to accelerate sequence alignment on the GPU, and I used the method to investigate whether the Operational Taxonomic Unit (OTU) classification problem can be improved with such sheer amount of computational power. I have developed a method to accelerate pairwise k-mer comparison on the GPU, and I used the method to further develop PolyHomology, a framework to scaffold shared sequence motifs across large numbers of genomes to illuminate the structure of the regulatory network in yeasts. The results suggest that such approach to heterogeneous computing could help to answer questions in biology and is a viable path to new discoveries in the present and the future.

Chapter 1

Introduction & Background

1.1 Introduction

1.1.1 The need for accelerated computing in bioinformatics

A number of problems in the field of bioinformatics, systems biology, and computational biology field require abstracting physical entities to mathematical or computational models [94]. In such studies, the computational paradigms often involve algorithms that can be solved by the Central Processing Unit (CPU). Historically, those algorithm benefits from both advancements of computing power in both performance gains in the serial processing capabilities of individual CPU cores and parallel gains - more cores per CPU and more computers per network. However, the increases in serial performance have slowed down over recent years [55], and scaling out CPU has shown to be cost-prohibitive [103]. More alarmingly, recent research has shown that approaches and practices used to accelerate the CPUs that are traditionally considered safe for decades are fundamentally insecure [73] [62].

Moreover, the interdependence between biology and mathematical modeling is well

known. The limitation of new discoveries in biology have shifted from the lack of data and equipment to lack of suitable models [22]. Massive quantities of sequencing data can be generated by platforms such as Illumina [25] [109], but tools to analyze such data have not kept pace with this increase in data volume. One approach to address this problem has been to explore acceleration of analyses using parallel computing tools such as clusters, cloud-computing, and heterogeneous accelerator platforms such as the Intel Many Integrated Core (MIC) Architecture, Field-Programmable Gate Array (FPGA) devices, GPUs, or hybrid solutions such as Accelerated Processing Units (APUs) [152].

While scaling-up and scaling-out approaches (obtaining more powerful silicon) are popular, they are more cost-prohibitive and are approaching a plateau [103]. Moreover, the energy envelop required to operate an exascale cluster comprising about 100,000 compute cores is essentially prohibitive with current CPU technology [152]. Thus, heterogeneous platforms are being actively researched as an alternative in medium-sized research institutes. In the GPU field, the de-facto platform of choice is from NVIDIA corporation, which uses the CUDA libraries and has gained attention as complementing or replacing traditional CPU approaches.

Heterogeneous accelerator platforms enable research projects in bioinformatics in two main themes [121]:

- Utilizing new parallel optimizations ("porting") of well-known algorithms to existing pipelines and programs [74] [165] [112] [78].

- Deriving new algorithms to enable new research directions and paradigms. This includes innovations such as enabling demanding visualizations [134] [135] and simulations [45].

Traditionally, because computational power was limited, thus there was always a trade-off between computational power and the level of detail that the models take

in into consideration. This research tackles those limitations by the applying the increased computational power provided by the accelerator platforms to classically intractable problems. Our hypothesis is that new patterns emerge as more data comes in, thus as long as the algorithm can handle these increased data flow, we could answer and improve the answer by processing the additional data.

1.2 Organization of this work

The premise of this research is to investigate the applicability of various parallel computing platforms to several problems in the detection and analysis of homology in biological sequence. I hypothesize that by exploiting the sheer amount of computation power and sequencing data, it is possible to deduce information from raw sequences without supplying the underlying prior knowledge to come up with an answer. I have developed such tools to perform analysis at scales that are traditionally unattainable with general-purpose CPU platforms. I have studied both the algorithms, their respective performance in the GPU context, and investigated biological questions with the tools I have developed.

The basic algorithms investigated with complete, exhaustive search algorithms that are unattainable on the CPU are:

- GPU Parallel Needleman-Wunsch alignment.
- GPU Parallel *k-mer* comparison.

The two aforementioned problems are addressed in Chapter 2 and Chapter 3 of this thesis, respectively. With those algorithms as a start, I addressed the following biological question:

- Does the Operational Taxonomic Unit (OTU) classification problem benefit from such an exhaustive approach?

- Can scaffolding of shared sequence motifs across large numbers of genomes illuminate the structure of the regulatory network?

To address those questions, the NVIDIA GPU platform is used to conduct pairwise analyses. Specifically, the following packages were developed:

- PolyHomology, a suite of algorithms and tools that mine motifs from the upstream regions of genes. The motif length is specified beforehand can range up to 30 nucleotides long. The suite accomplishes the goal by scaffolding homology information data from multiple yeast genomes.

- NEAT, a suite of pairwise comparison program to construct pairwise alignment graph for any given set of sequence, and apply to understand the OTU structure from the 16S genes database.

The answer to the biological questions powered by the two packages is presented in Chapter 4 and Chapter 5 of this thesis, respectively.

1.3 Background

1.3.1 The pairwise sequence alignment problem

The pairwise sequence alignment is one of the most basic operations in sequence analysis and is essential to many problems, including phylogenetics and molecular evolution studies. Each comparison is carried on by a dynamic programming procedure (using either the Needleman-Wunsch [88] or the Smith-Waterman [128] algorithm) in order to retrieve an alignment or similarity score between a pair of sequences. Each pairwise alignment problem for s sequences has a polynomial complexity of $O(s^2)$. Each individual pairwise comparison, in turn, has the complexity of $O(n^2)$ (where n is the length of the sequences). The multiplication of these two factors results in the

time needed for the comparison of many long sequences being computationally expensive. However, biologists require such comparisons to being handled quickly. In some cases, there are available heuristics given a degree of background knowledge about the underlying data. For instance, if relatively few of the sequences being compared bear significant similarities to each other but the vast majority do not, a heuristic can halt most comparisons and alignments when the dissimilarity passes a threshold, thus saving significant time.

However, there exists a class of problems where all the sequences bear a considerable similarity. This situation is true of metagenomics studies which employ the 16S rDNA gene. Metagenomics sampling from the environment enables researchers to study microbial communities that they are unable to cultivate in lab settings [81]. Because the 16S rDNA gene is an essential gene among prokaryotic species, it is highly conserved. Thus, orthologs from different species bear high similarities, making it impossible to apply such heuristic-pruning techniques. If one wishes to construct a "tree of life" using the 16S gene, it is necessary to compare every single pair of sequences. According to Wood et al. [162], programs that are used to analyze DNA libraries are "relatively slow and computationally expensive." With the continual improvements in next-generation sequencing techniques, for example, the cheap and abundant micro-Biome kit [28], the amount of data generated by metagenomics studies is increasing rapidly. Thus, the need for a fast algorithm that can run on a standard PC is needed, not only for traditional researchers but also for enabling citizens' involvement in science [150].

There have been several solutions proposed for the alignment problem. On the GPU, the solution proposed by Manavski-Valle provides up to 30 times speedup [80] with local alignment. However, this solution is not applicable to our global alignment problem. With global alignment, the reference benchmark suite Rodinia provides up to 8x speedup over one CPU core [19]. However, Rodinia relies heavily on the GPU

global memory access; thus the solution does not scale well and provides diminishing returns compared to a multi-threaded CPU version [19].

In a previous study, we have shown that in the special case when only the identity score is required and the actual alignment itself is not considered, then an 8x speedup is achievable on a workstation-grade GPUs [68]. Recently, Balhaf et al. [7] proposed a method that provides an 11x speed up to the Levenstein distance calculation by implementing a diagonal-based tracing technique.

None of the aforementioned solutions are designed for exhaustive pairwise comparison. Moreover, none take into account possible heuristics that could be employed to further reduce the number of computation steps needed: instead, they always carry out a full distance calculation.

1.3.2 The Needleman-Wunsch Global Alignment Algorithm

The classic Needleman-Wunsch algorithm uses a dynamic programming approach to calculate the cost function to align two arbitrary sequences of length m and n. The cost function is a measurement of the relative similarity between two sequences and is computed by a list of successive operations. The operation is either an insertion/deletion of a base or an alignment of two bases (one from each sequence) in the form of a match or mismatch. In the case of protein sequences, the cost of a mismatch is defined by a scoring matrix that is configurable at run-time S. For example, the BLOSUM62 scoring matrix scores the log likelihood of the two amino acids in question being homologous [33]. Hence, the total score is proportional to the probability that two sequences being homologous. In the case of DNA protein sequences in the aforementioned problem, a simple, static scoring scheme can be used. Note also that the introduction of a new gap by an insertion/deletion operation has a relatively high fixed penalty. In some implementations, the extension of an already open gap adds a smaller penalty. To simplify the problem, here we use a constant gap cost G.

From the description, each of the comparisons requires $O(nm)$ space and $O(nm)$ time to compute (often simplified as $O(n^2)$ when given the two sequences have the same length $m = n$). Hirschberg's algorithm can reduce the space complexity to $O(n)$ [50]. For a database of s sequences, the complexity of all pairwise comparisons is $O(ns^2)$ for space and $O(n^2s^2)$ for time requirement. Our database, consisting of the genes from The Ribosomal Database Project [100] and NCBI GenBank [11], provides 25,000 unique 16S rDNA sequences of bacteria: this complexity implies a total of 300 million comparisons. With each comparison being a quadratic time complexity operation, the CPU is an unfeasible option.

1.3.3 K-mer counting and motif finding problem

Discovering unknown sequences of certain lengths is a valuable tool for biologists, and is applicable to applications such as motifs discovery. Motifs are short sequences in DNA, most often upstream of the coding portion of a gene, that regulate the gene by the binding of special transcription factor (TF) proteins [35] [84]. These TFs in turn control whether or not the gene is expressed as a mRNA and potentially as a protein. A small fraction of expected set of motifs in various organisms are known and have relatively standard and simple sequences, for example, the TATA-box region. However, the rest are unknown: for instance, 50% of all genes in the human genome are estimated to have alternative promoters [13] [26]. To add to the complexity, while a single transcription factor generally recognizes a single motif, that motif can be degenerate, meaning that in some positions multiple bases are tolerated. Coupled with the fact that one gene may well be regulated by several TFs, this problem can be challenging [48].

Thus, a *de-novo* algorithm to dynamically account and mine all variants of any short sequence of length k ($k-mer$) that can potentially be a motif is needed. While sequence alignment algorithms such as Needleman-Wunsch and Smith-Waterman are

effective in comparing two known sequences, they are not applicable when searching for unknown sequences. Space and time complexity are among the reasons why an alignment algorithm is ineffective [91]. First, representing each kmer as a sequence has the "curse of dimensionality" phenomenon: the space needed for such operation exponentially increases with the length of the sequence k. An $8 - mer$ requires $4^8 = 65,536$ vectors to account for all possible 8-mers, but adding just two bases increases the size of the vector to over one million. Second, deducing the distance between any pair of $k - mers$ using the Needleman-Wunsch algorithm is a quadratic operation in terms of time complexity. To solve this problem, software packages often use a statistical model combined with prior knowledge to estimate the significance of the motifs in question [92] [163].

The advent of inexpensive DNA sequencing technologies [125] means that a many genomes of closely-related species are becoming available, for instance among the fruit flies [20], yeasts [16], and mammals [38]. With such data, it is possible to use a comparative approach to this problem. We propose a new method to identify clusters of co-regulated genes, by analyzing the similarity of upstream sequences across dozens of genomes.

1.3.4 The NVIDIA CUDA platform

The CUDA platform is a scalable parallel computing model designed by the NVIDIA corporation for high-performance computing applications [93]. The programming model uses the GPU as an accelerator to the ordinary CPU when an intensive computing task is required. For the task to be accelerated, it first has to be expressed as a parallelizable problem. Then the algorithm has to be written in a way that utilizes the NVIDIA architecture memory and computational hierarchy.

With regard to the computational hierarchy, the GPU consists of several streaming processors (SMs), and each streaming processor contains tens to hundreds of smaller

scalar cores. When executing, a set of at least 32 cores executes the same instruction and is called a warp. On the programming interface, the logic of the hardware is exposed to the programmer as threads (code executed on individual cores), blocks (groups of warps), and grid (the whole GPU). The architecture is different from the CPU in two fundamental ways. First, the cores need to have very limited capabilities in order to keep them small and efficient. Second, they are structured in a Single-Instruction, Multiple-Data (SIMD) architecture, meaning that all of the cores in the same warp have to execute the same instruction on different pieces of data, instead of being the independent units seen in the CPU case.

NVIDIA GPUs have a two-level memory hierarchy. First, there is the off-chip, slow, high-latency read-write memory, often referred to as the global memory. Second, there is a small on-chip, fast read-write memory often referred to as the shared memory or scratchpad. Besides the shared memory, an off-chip, fast read-only memory can also be used for special purposes. In traditional CPU models, the CPU generally takes care of the caching of data and instructions. However, in the NVIDIA CUDA platform, the programmer has the power to designate at a much finer level how the cache behaves.

With those considered requirements in mind, a modern consumer NVIDIA GPU can provide a speedup of 2x-40x compared to the CPU for sequence alignment, molecular dynamics, protein docking and other applications in bioinformatics [94]. Combined with a familiar C/C++ programming base that is nearly identical to CPU, the GPU hence provides a good trade-off between speedup, development time and cost.

Chapter 2

Large-Scale Pairwise Alignments on NVIDIA GPUs

This chapter has previously appeared in the substantially the same form as: Huan Truong et al. "Large-scale pairwise alignments on GPU clusters: exploring the implementation space". In: *Journal of Signal Processing Systems* 77.1-2 (2014), pp. 131–149. I contributed to this research in the LazyRScan algorithm, design optimization, test data collection and filtering, and benchmark. The computing cluster setup part has been omitted from the original paper as it is not relevant to the scope of this dissertation.

2.1 Introduction

The pairwise sequence alignment algorithms, both local and global [89] [129], are in many ways the core technology for the study of biological sequences. They have key roles in multiple sequence alignment [142], phylogenetics [49], and molecular evolution studies [90]. In addition, heuristic improvements to the basic dynamic programming approach are essential features of sequence database search programs such as FASTA [106] and BLAST [4, 5] and various forms of genome assembly algorithms [64] [69]. Such acceleration is useful because, while the dynamic programming approach to alignment is only $O(n^2)$ in time complexity, biologists often wish to make millions or even billions of such comparisons [10]. However, these heuristics depend on the assumption that the vast majority of the sequence pairs being compared have essentially no similarity and that, once this fact has been demonstrated for a sequence pair, the computation of the alignment itself is unnecessary.

Increasingly a second class of problem is becoming relevant. In this case, there is a requirement to compare very large numbers of sequences that are all evolutionarily related. As a result, it is not possible to omit the computation of any of the alignments, making approaches such as that of BLAST inappropriate. One example is the computation of very large multiple sequence alignments for analyses such as inference of the "tree of life" [85, 101, 104]. A similar problem motivates our work here, namely the analysis of complex microbial communities through the sequencing of a particular microbial gene, the 16S rDNA gene. Biologists have discovered that many microbes cannot be cultured under laboratory conditions but that it is possible to assess their presence through the direct sequencing of the DNA in an environment [9, 61, 151, 159]. To compare microbial communities across environments, it is helpful to survey a single gene: the 16S gene is useful in this regard as it is essentially ubiquitous across prokaryotic life. However, the sequencing of the gene is only a first step: it is then necessary to compare the sequences generated to each other and to

other known 16S sequences to assess the taxonomic diversity present in the sample. As there are hundreds of thousands of 16S sequences in sequence databases and tens of thousands of unique sequences among those [23], this analysis can be daunting.

The problem as stated is clearly highly parallel, and, as such, we sought to bring the massively parallel computing potential of GPUs to bear on it. General-purpose graphics processing units (GPGPUs) are advancements of hardware originally developed to accelerate complex graphical rendering for applications such as 3D gaming. These devices can be programmed in several ways, including the CUDA framework proposed by Nvidia. The design of parallel kernels for GPU directly affects the utilization of the underlying hardware: its compute cores and memory hierarchy. This usage, in turn, influences the performance achieved.

Our contributions can be summarized as follows.

- We propose four implementations of multiple pairwise alignments using the Needleman-Wunsch (NW) algorithm on GPU. Three of our parallel kernels (TiledDScan-mNW, DScan-mNW and RScan-mNW) are general purpose. Our forth implementation (LazyRScan-mNW) is optimized for problems that require performing the trace-back operation only on a subset of the sequence pairs in the initial dataset (for example, the pairs which alignment score exceeds a predefined threshold).

- The methods considered differ in their computational patterns, their use of the available hardware parallelism, and their handling of the data dependences intrinsic in NW. Our analyses give insights into the architectural benefits and costs of using GPUs for bioinformatics, insights likely applicable to other domains.

- We evaluate our framework on a dataset of about 25,000 unique 16S rDNA genes from the Ribosomal Database [23]. We use a variety of Nvidia GPUs: from the

low-end Quadro 2000 to the high-end Tesla K20. We show that our optimizations are effective on all of the considered devices. Our experiments based on the general purpose TiledDScan-mNW, DScan-mNW and RScan-mNW kernels show a throughput in the order of 250 and 330 pairwise alignments/sec on low- and high-end GPUs, respectively. In addition, we achieve a throughput of 1,015 pairwise alignments/sec on a 6-node commodity cluster equipped with a low-end GPU on each node. Finally, our LazyRScan-mNW kernel allows throughputs up to about 4,000 and 6,000 pairwise alignments/sec on low- and high-end GPUs, respectively.

2.2 Background

2.2.1 Related Work

In recent years GPUs and other accelerator devices have been widely used to accelerate a variety of scientific applications from many domains [96, 18, 139]. In particular, a number of biological applications, including BLAST [153], hidden Markov models [118, 155] and structure comparisons [102], have been ported to GPU or FPGA architectures. Most relevant to our work are several sequence alignment algorithms implemented on GPU [18, 76]. In Section 2.2.3, we will provide more background on one of these: the NW implementation in the Rodinia benchmark suite [18].

Among the alignment implementations, Liu et al. [77] present an optimized sequence database search tool based on the Smith-Waterman (SW) local alignment algorithm (in contrast to the NW global alignment problem considered here). Compared to other implementations [138, 160], their tool provides better performance guarantees for protein database searches. Li et al., [70] offer a GPU acceleration of SW intended for a single comparison of two very long sequences; we focus on accel-

erating many pairwise alignments of shorter sequences. We are interested in the NW problem, which rather than being used for database search is more commonly applied to situations where all possible pairwise alignments are required (e.g., alignments for phylogenetics or metagenomics as described above). In their first phase (computation of the alignment matrix), NW and SW share similar computation patterns, so optimization techniques can be reused between the two methods.

There are also distributed CPU-based implementations of NW: for example ClustalW-MPI [71] aligns multiple protein, RNA or DNA sequences in parallel using MPI. Biegert et al., [12] have introduced a more general MPI bioinformatics toolkit in the form of an interactive web service that supports searches, multiple alignments and structure prediction. Our tool differs from these in combining MPI and CUDA to allow deployment on CPU/GPU clusters where multiple GPUs may be employed simultaneously.

2.2.2 Parallelism in NVIDIA GPUs

Nvidia GPUs comprise a set of Streaming Multiprocessors (SMs), where each SM in turn contains a set of simple in-order cores. These in-order cores execute instructions in a SIMD manner. GPUs have a heterogeneous memory organization consisting of high latency off-chip global memory, low latency read-only constant memory (which resides off-chip but is cached), low-latency on-chip read-write shared memory, and texture memory. GPUs adopting the Fermi and Kepler architecture, such as those used in this work, are also equipped with a two-level cache hierarchy. Judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieve good performances. In particular, the utilization of the memory bandwidth can be optimized by performing regular access patterns to global memory. In this situation, distinct memory accesses are automatically coalesced into a single memory transaction, thus limiting the memory bandwidth used.

The advent of CUDA has greatly simplified the programmability of GPUs. With CUDA, the computation is organized in a hierarchical fashion, wherein threads are grouped into thread-blocks. Each thread-block is mapped onto a different SM, whereas different threads are mapped to simple cores and executed in SIMD units, called warps. The presence of control-flow divergence within warps can decrease the GPU utilization and badly affect the performance. Threads within the same block can communicate using shared memory, whereas threads within different thread-blocks are fully independent. Therefore, CUDA exposes to the programmer two degrees of parallelism: fine-grained parallelism within a thread-block and coarse-grained parallelism across multiple thread-blocks.

2.2.3 Rodinia-NW: Needleman-Wunsch on GPU

The Rodinia benchmark suite [18] offers a GPU parallelization of Needleman-Wunsch algorithm (hereafter, Rodinia-NW), that we will use as baseline.

Rodinia-NW operates as follows. Since each element in the alignment matrix depends on its left-, upper- and left-upper-neighbors, a way to exploit parallelism is by processing the matrix in minor diagonal manner. Each minor diagonal depends on the previous one, thus leading to the need for iterating over minor diagonals. However, at every iteration, all the (independent) elements in the same minor diagonal line can be calculated simultaneously. If the matrix is laid out in global memory in row-major order, the involved memory access patterns are uncoalesced, potentially leading to performance degradation. Since each element in the alignment matrix is used for calculating three other elements, performance can be improved by leveraging shared memory and dividing the alignment matrix in square tiles (each of them fitting the shared memory capacity). Rodinia-NW performs tiling and exploits two levels of parallelism: (i) within each tile elements are processed in minor diagonal manner, and (ii) different tiles in the same minor diagonal line can also be processed concurrently

by distinct thread-blocks. Threads within the same thread-block manipulate the data and store elements in shared memory temporarily. After the computation of a tile completes, all of the data are moved to global memory using coalesced accesses. For square alignment matrices and tiles of width N and T, respectively, Rodinia-NWs parallel kernel is invoked $2 \times (N/T) - 1$ times (once for each minor diagonal of tiles). After carefully analyzing Rodinia-NW, we found the following limitations.

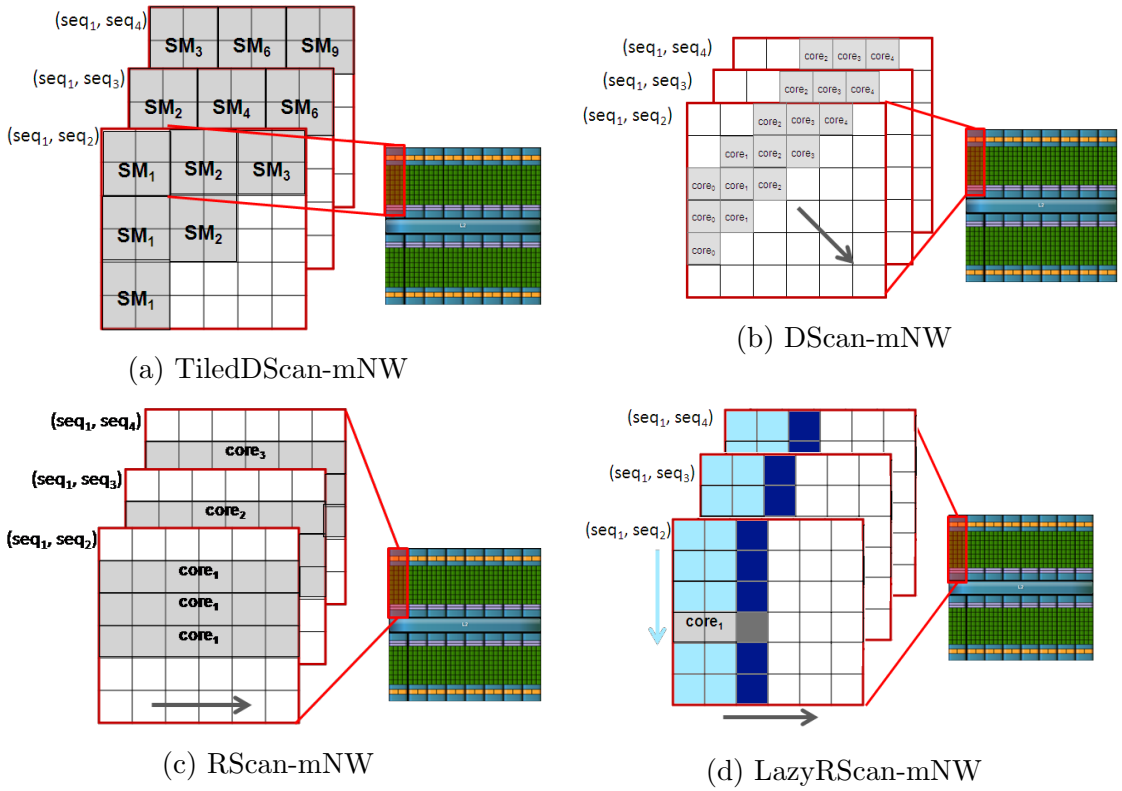
First, Rodinia-NW is designed for a single pairwise comparison. Applications such as those above require hundreds to thousands of comparisons. As such, they introduce a second exploitable level of parallelism, especially as each pairwise comparison is independent. Moreover, the sequences generally differ in length but Rodinia-NW only supports sequences of equal length, requiring padding to handle more general cases.

Second, Rodinia-NW requires three data transfers for each alignment, an approach that can be improved. Before kernel launch, the alignment matrix is initialized (with the gap information) on the CPU. Next, alignment matrix and score matrix are copied from CPU to GPU. The alignment matrix is processed on the GPU, and finally copied back to the CPU. We note that the two copies of the alignment matrix are $O(nm)$ each. However, the first data transfer of the alignment matrix can be avoided by initializing its 1^{st} row and 1^{st} column directly on the GPU.

Finally, CUDA does not support global barrier synchronization among thread-blocks within a parallel kernel (an implicit global synchronization takes place at the end of each kernel execution). Since in Rodinia-NW each tile is mapped to a thread-block and tiles must be processed in diagonal strip manner, a global synchronization among thread-blocks operating on the same diagonal is required before proceeding to the next diagonal. This is accomplished by invoking multiple kernel launches from the host side. This approach has two limitations: (i) each kernel launch has an associated overhead (that depends on the GPU device), and (ii) the GPU is underutilized by

kernel launches that process small numbers of tiles (i.e., those corresponding to the first and the last diagonals).

2.3 Design of the algorithms



(a) TiledDScan-mNW

(b) DScan-mNW

(c) RScan-mNW

(d) LazyRScan-mNW

Figure 2.1: Exemplification of our 4 GPU implementations of NW-based parallel pairwise alignments and of the mapping to the underlying GPU cores and SMs. In TiledDScan-mNW, the alignment matrices are tiled, and each tile is processed by a thread-block (and mapped onto a SM). In DScan-mNW, every alignment matrix is processed by a thread-block (and mapped onto a SM). In RScan-mNW and LazyRScan-mNW, every alignment matrix is processed by a thread (and mapped onto a core). However, in LazyRScan-mNW, every thread only writes to the global memory the last column of each slice (denoted by darker blocks).

In this Section we describe four alternative implementations of multiple pairwise alignments using NW on GPUs: *TiledDScan-mNW, DScan-mNW, RScan-mNW* and *LazyRScan-mNW*. All these implementations, exemplified in *Figure* 2.1, aim to overcome the limitations pointed out above.

TiledDScan-mNW: Multiple alignments with tiling

The first method (TiledDScan-mNW) is a directed extension of Rodinia-NW to multiple pairwise alignments. This approach still uses tiling and operates in diagonal strip manner, performing multiple kernel invocations to compute the alignment matrices. However, for each kernel invocation, multiple alignment matrices are concurrently processed using different thread-blocks (and SMs). This is illustrated in *Figure* 2.1*a*, where we concurrently perform three pairwise comparisons: *(seq1, seq2), (seq1, seq3) and (seq1, seq4).* In the first iteration, the top-left tiles of the three matrices are processed in parallel by three thread-blocks, and thus mapped onto three streaming multiprocessors: SM1, SM2 and SM3. In the second iteration, the tiles of the second minor diagonal of the three matrices are processed in parallel by six thread-blocks, and thus mapped onto streaming multiprocessors SM1-SM6. Note that, for m pairwise comparisons, the number of kernel invocations of TiledDScan-mNW is reduced by a factor m (as compared to Rodinia-NW); for each kernel call, the number of thread-blocks is increased by a factor m. This has two advantages: (i) a limited kernel invocation overhead, and (ii) an improved GPU utilization. Execution configurations with a large number of threads allow not only exploiting all the SMs and cores available on the GPU, but also hiding the global memory access latencies (and NW is a memory-intensive application). Being an extension of Rodinia-NW, TiledDScan-mNW retains its advantages: regular computational patterns and coalesced memory access patterns when storing alignment data from shared memory to global memory.

DScan-mNW: Single-kernel diagonal scan

TiledDScan-mNW still requires multiple kernel invocations to perform m pairwise alignments. Even if the parallelism within each kernel call is improved by a factor m compared with Rodinia-NW, some kernel invocations still exhibit limited parallelism

(and limited opportunity to hide memory latencies). Our second implementation DScan-mNW performs a diagonal scan with a single kernel call. As illustrated in *Figure* 2.1*b*, in this case each alignment matrix is assigned to a thread-block (and mapped onto a SM). No tiling is performed. The computation iterates over diagonals. For each diagonal, every element is processed by a thread (and mapped onto a core). To limit the number of expensive accesses to global memory, the computation is fully performed in shared memory. The alignment matrix is stored in row-major order in global memory and in minor diagonal order in shared memory. According to equation (1), at each iteration three diagonal lines are required: the first two diagonal lines cache previous data and the third one contains the newly computed elements. Once computed, this third line can be copied from shared to global memory. At that point, the first diagonal line can be discarded and the shared memory reused for the next iteration. To summarize, the matrices are created in shared memory and moved to global memory diagonally. The main disadvantage of this approach is the uncoalesced memory accesses required to store diagonal data to global memory. We found that the latencies of such irregular access patterns can be effectively hidden by using large numbers of threads.

The computational pattern of our DScan-mNW is similar to the SW intra-task parallelization proposed by Liu et al. [77]. However, [77] avoids uncoalesced memory accesses by storing the alignment matrix in global memory in minor diagonal order. We found that, when using large thread-blocks to hide memory latencies (e.g., 512 threads/block), the overhead due to uncoalesced memory access patterns is reduced to 10% and 7% of the execution time on Fermi and Kepler GPUs, respectively (the exact percentage depends also on the clock-rate of the memory system). On the other hand, storing the alignment matrix in row-major facilitates the trace-back operation (which is not considered in [77]) in two ways: first, it avoids the need for complex index translation; second, the more regular data layout leads to better caching properties.

RScan-mNW: Row scan via single CUDA core

Our third method RScan-mNW uses a *fine-grained matrix-to-core mapping and a row-scan* approach. First, each alignment matrix is computed by a single GPU core. Second, to allow regular compute and memory access patterns, each alignment matrix is computed row-wise (rather than diagonal-wise). This computational pattern is illustrated in *Figure* 2.1*c*.

This method leverages shared memory in order to allow data reuse and minimize the global memory transactions. The parallel kernel iterates over the rows of the alignment matrices. For each iteration, only two rows per matrix must reside in shared memory: the previously computed one and the one containing newly computed elements. Only the left-most element of the new row must be loaded from global memory; for the rest, the computation happens solely in shared memory. Once the new row has been computed, it is copied from shared to global memory. The previously computed row can be discarded, and the new one can be cached for use in the next iteration. The kernel has two phases: computation and communication. In the computation phase, the threads within a thread-block operate fully independently: each thread computes the data corresponding to the row of an alignment matrix and stores them in shared memory. In the communication phase, threads belonging to the same thread-block cooperate to transfer row data from shared to global memory in a coalesced fashion (that is, each alignment matrix is transferred cooperatively by multiple threads). In case of very long sequences, rows are split into slices so as to fit into shared memory. The size k of these slices is configurable. Large slices require more shared memory, which in turn limits the number of active threads on each SM. Small slices (e.g. $k < 32$ elements) lead to warp underutilization in the communication phase, which in turn can hurt the performance. The usage of shared memory is a major concern in the kernel configuration process. The per-block shared memory can be calculated using the following formula:

$shmem = BLOCK\_SIZE \times k \times (1char + 2ints)$

Each thread stores three sets of data: the sequence data and two sections of the alignment matrix. Each thread-block performs $BLOCK\_SIZE$ pairwise alignments using slices of size k. By setting the $BLOCK\_SIZE$ and k to 32, we use 12KB of shared memory with no warp underutilization. With this setting, each SM can concurrently run up to four thread-blocks.

The advantages of this approach are twofold. First, the computational pattern is extremely regular: unlike diagonals, rows are all of the same size. Second, data transfers between shared and global memory are naturally coalesced. The main drawback to this approach is that the parallelism is limited by the GPU memory capacity. For example, if the sequences to be compared are of length 2,000 and the alignment matrices contain 4-byte integers, then each matrix will be of size 32MB. To fully utilize the cores of typical GPUs (say 480 cores), we should allow 480 parallel pairwise comparisons, requiring a total of roughly 15GB of memory. This number considerably exceeds the 1-5GB of memory present on most GPUs. Therefore, on long sequences RScan-mNW will tend to underutilize the GPU resources. On the other hand, this approach is very promising for short sequences (e.g. < 500). For long sequences, an alternative optimization would be to break the alignment matrices into smaller strips to reduce the memory footprint, and use dual-buffering to move previously computed strips to the CPU while computing new ones. Finally, we note that certain scoring schemes allow for linear memory NW algorithms of minimal complexity: under these limited and less-commonly used schemes, highly efficient parallelism could be achieved using RScan-mNW.

The computational pattern of our RScan-mNW is similar to the SW inter-task parallelization proposed by Liu et al. [77]. However, their proposal does not use shared memory in the kernel and adopts a different data layout in global memory. Specifically, to avoid uncoalesced global memory accesses, Liu et al. place data cor-

responding to different alignment matrices into continuous global memory space. For instance, the i^{th} element of global memory is from the i^{th} alignment matrix, while the $(i+1)^{th}$ element is from the $(i+1)^{th}$ alignment matrix. This memory layout leads to poor data locality during the trace-back phase. As mentioned above, trace-back is not considered in [77], but is a necessary operation in the problem we consider.

LazyRScan-mNW: Whole matrix calculation via single CUDA core with lowered global memory requirements

```
1: Initialize matrix size (w, l)
2: for each slice size (k, l):
3:     calculate row scores for row 0
4:     for row ← 1 to l:
5:         load leftmost score for column 0 from global memory
6:         for column ← 1 to k:
7:             calculate value for(row, column) in shared memory
8:         save rightmost score for column l to global memory
9:      end for row
10:    advance slice by k
11: end for each slice
11: returns final matrix score
```

Figure 2.2: Pseudo-code version of LazyRScan-mNW.

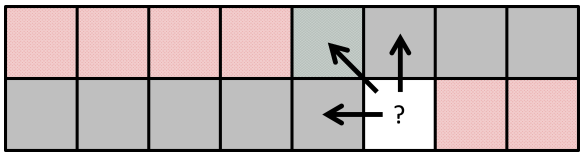


Figure 2.3: Storing in shared memory a hybrid row consisting of cells belonging to two contiguous rows allows saving half of the shared memory required for row calculation. As computation (white cell) progresses from left-to-right horizontally, obsolete cells (cross-hatched in red) are gradually evicted from shared memory. To complete the computation, only the solid grey cells as well as an interim diagonal look-back value (denoted with the zig-zag pattern) are required. The diagonal loopback value is stored in a register.

The three aforementioned methods share the same problem: the maximum number of alignments that can be computed in parallel is limited by the amount of global memory. As discussed, this limitation is particularly significant in the case

of $RScan - mNW$, where each alignment is mapped onto a GPU core. In this case, the limited number of alignment matrices that can be accommodated in the available global memory leads to severe core underutilization. However, several applications only require the actual sequence alignment (computed through the trace-back operation) only for a subset of the pairs in the dataset. For these applications, it is possible to record only the alignment score (rather than the whole alignment matrix). This score can then be used as a filter to avoid performing unnecessary trace-back operations on sequence pairs that are too dissimilar. Exactly such a requirement spawned our initial interest in this topic namely computing all possible pairwise comparisons among a large number of sequences but retaining only those with a high level of similarity. In this case, the lack of the need to have the whole alignment matrix stored in global memory makes it possible to derive a method that can make full use of the computational power of the GPU cores.

With this relaxed requirement in mind, $LazyRScan - mNW$ optimizes our row-scan approach (Rscan-mNW) so as to minimize global memory reads and writes and at the same time make better use of the shared memory available on the streaming multi-processors. The new method is based on the following insight: much of the information in one slice of an alignment matrix can be discarded safely during the computation of that slice. LazyRScan-mNW is extremely frugal about global memory usage, and avoids accessing global memory whenever possible. The computational pattern is illustrated in Figure 2.1d and presented in the pseudo-code in Figure 2.2.

Similarly to RScan-mNW, in order to effectively use shared memory even in the presence of long sequences, LazyRScan-mNW divides the alignment matrix into vertical *slices* of k columns each. However, when copying the scores into global memory, this method only retains the scores in the last column of each slice (and discards the other scores). For each matrix, the computation of the next slice resumes where the last one left off, by taking the stored rightmost scores of the last computation as its

leftmost columns scores. Given the leftmost column's scores, each iteration of vertical slice can compute the rightmost column's scores by performing the standard NW computation. Thus, by carefully selecting k (see section 2.5.3), we reduce the number of write operations to global memory. Computing a slice of width k and height l requires 2l global memory operations (l reads and l writes). In addition, for each matrix, the global memory requirement is limited to l scores, and the shared memory requirement is linear in k. Limiting the shared memory utilization allows more thread-blocks to be executed in parallel, thus hiding global memory access latencies.

We further reduce the shared memory usage of each vertical slice by implementing an optimization suggested by Myers and Miller [87]. Specifically, instead of storing in shared memory two distinct rows (the current and the previously computed one), we store a hybrid row obtained by progressively overwriting the last computed row with the current one. This method requires recording an additional interim *diagonal look-back value*, representing the conflicted array element that would otherwise be overwritten by the next row's value. We store this additional value in a register. Figure 2.3 illustrates the optimization.

LazyRScan-mNW inherits all the advantages of RScan-mNW as well as some additional benefits. First, each thread only requires a linear amount of global memory. Thus, the global memory required will be equal to $batch\_size \times l$ integers, where l is the length of the longest sequence, and $batch\_size$ the number of pairwise alignments computed in parallel. Second, instead of reading and writing to global memory at every cell, the method reads and writes to global memory once every k cells. This allows the GPU scheduler to alternate between warps that are performing calculation and warps that are waiting on memory accesses, thus better hiding the latency between global memory accesses. Third, the amount of shared memory required to store the scores is halved using the optimization described above.

However, this method has a drawback: it is no longer trivial to restore the align-

Method	Global memory	Per-block shared memory
Rodinia-NW	$batch\_size \times 2l^2$ ints	$2k^2$ ints
TiledDScan-mNW	$batch\_size \times l^2$ ints	$k^2 ints + 2k$ chars
DScan-mNW		$3l$ ints + $2k$ chars
RScan-mNW		($2k$ ints + k chars) $\times block\_size$
LazyRScan-mNW	$batch\_size \times ints$	(k ints + k chars) $\times block\_size$

Table 2.1: Global and per-block shared memory requirements for different methods (the amount of storage needed for actual sequence data is not accounted for in global memory usage). A batch is a set of pairwise alignments processed in parallel on the GPU.

ment matrix, making the trace-back operation challenging. LazyRScan-mNW is therefore more suitable to applications that require the trace-back to be calculated only for a subset of the sequence pairs (for example, those with a similarity score above a predefined threshold).

Comparisons of memory usage between different methods

Table 2.1 summarizes the global and per-block shared memory usage of the described methods. In the table, *batch_size* is the number of pairwise alignments computed in parallel, l is the length of the longest sequence, and k denotes the slice size, and is a configurable value smaller than l.

2.4 Design optimizations

Pinned memory - Generally there are three stages on a GPGPU computation. First, the initial data will be copied from the host to the device's global memory. Then, the CPU will launch the kernel, allowing the calculation of the results on the device. After this operation finishes, the data will be copied back to the host memory. With pageable memory, the memory copy operations contribute a significant fraction of the time of the GPGPU computation due to the mandatory involvement of the

CPU, as memory allocated on the host may not be physically present and thus may require swapping. Pinned memory guarantees that the memory allocated is always physically present in the host's physical RAM. Pinned memory therefore can provide significant speedups to device-and-host memory transport, as the GPU can handle the transfer using its DMA hardware capabilities without the involvement of the CPU. This approach has two potential advantages. First, the copy operation on pinned memory is usually much faster than for pageable memory [115]. Second, the CPU only needs to pass the list of physical pages to the driver and is then free to do other work.

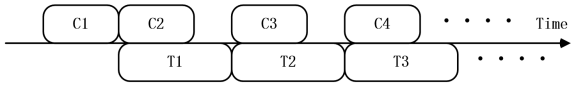


Figure 2.4: Exemplification of dual-buffering.

Dual-buffering - Dual-buffering is a common scheme for reducing overhead by overlapping communication and computation. It requires that a non-blocking communication mechanism is provided by the system. In our system, there are two main operations that are time-consuming but can be executed concurrently: kernel calls to calculate the alignment matrix, and the memory copy operation to transfer the calculated alignment matrix from device's global memory back to the host's memory. Dual-buffering is employed in order to ensure mutual exclusion in the access to the two alignment matrices, one of which is being calculated and one copied. The result is that the kernel always works on half of the available memory, leaving the other half the buffer that is already fully calculated free to be transferred back in a non-blocking manner. By using dual-buffering, we can further exploit the concurrency offered by the GPU, by interleaving the two operations (Figure 2.4). To implement dual-buffering, we used asynchronous pinned memory operations and CUDA streams.

Global memory access pattern optimization Parallel threads in CUDA are grouped into thread-blocks, and the GPU coalesces the global accesses into as few

transactions as possible. Misaligned memory operations cause delays in the computation as some threads have to wait for the misaligned reads to complete. This issue however, can potentially be minimized. In the *LazyRScan-mNW* implementation, global memory reads and writes on the same position on different matrices of consecutive threads are also consecutive, enabling the hardware to perform memory access operations in a coalesced fashion.

Caching global memory reads into shared memory Global memory is needed to store the sequences, however reading the sequences from global memory is an expensive operation. In *(Lazy)RScan-mNW*, each slice of sequence of width k has a relatively small number of bases that need to be repeatedly accessed and compared against the other sequence. Thus, we avoided reading the slice sequence from global memory by caching the whole slice up front in shared memory. Given that, in the inner loop (Figure 2.2, line 6), each base of the slice is then repeatedly compared to one base from the other sequence, the base belonging to the other sequence could also be cached up front to reduce global memory read latency.

Device-to-host data-level optimization In the case of SlicedRScan-mNW, it is possible to transfer only the final score instead of the whole alignment matrix, largely eliminating the device-to-host transfer latency. In order to allow this simplification, we derive a lower bound L on the potential alignment score with a given percent identity. As an aside, the percent identity statistic is a reasonable indicator of sequence relatedness and is commonly employed in the biological literature. Our bound guarantees that all pairs having identity score at least as high as our threshold level are realigned and the trace-back computed on the CPU. Given identity score target M_{ID} $(0 < M_{ID} < 1)$ and two sequences of length m and n (where $m > n$), L is given by:

$L = m \times M_{ID} \times S_{match} \times 2 \times m \times S_{gap} \times (1 M_{ID})$

S_{match} is the alignment score for two matching bases and S_{gap} is the penalty for

GPU Type	Type	Values
Low-end GPU	Quadro 2000	4 SM x 48 cores 1 GB Global memory
Low-end GPU	GTX 460	7 SM x 48 cores 1 GB Global memory
Low-end GPU	GTX 480	15 SM x 32 cores 1.5 GB Global memory
High-end GPU	Tesla C2050	14 SM x 32 cores 2.6 GB Global memory
High-end GPU	Tesla C2070/C2075	14 SM x 32 cores 5 GB Global memory
High-end GPU	Tesla K20	13 SM x 192 cores 4.7 GB Global memory

Table 2.2: Characteristics of the GPUs used in our evaluation.

a gap (recall we are using linear gaps). For the required percentage identity is to be achieved, we must have at least $m \times M_{ID}$ positions that match. The worst case of an alignment with such a percent identity is then that all of the remaining positions are gaps (S_{gap}), giving us our value of L.

2.5 Experimental setup and evaluation

2.5.1 Experimental setup

Hardware setup Single GPU experiments have been performed on a variety of low-end and high-end GPUs, listed in Table 2.2.

Software setup The CUDA 5.0 driver and runtime are installed in all the machines used. The OS in use on the high-end cluster is CentOS5.5/6 with g++4.1.2; the OS in use on the low-end cluster is Ubuntu 12.04 with g++ 4.6.3. We used MPICH2 (version 1.4.1p1) as the implementation of MPI. Each data point represents the average across 3 executions.

Dataset Our reference dataset consists of about 25,000 unique 16S rDNA genes

from the Ribosomal Database [23]. The sequences are on average 1,536 bases long.

2.5.2 Performance on single GPU for general use cases

Our first set of experiments is meant to evaluate our GPU implementations and compare them with Rodinia-NW. In Section 2.2.3 we noted two limitations in Rodinia-NW: unnecessary memory transfers from CPU to GPU and inefficiencies in the computational kernel and its invocations. Below, we will show how we improve performance with respect to both limitations.

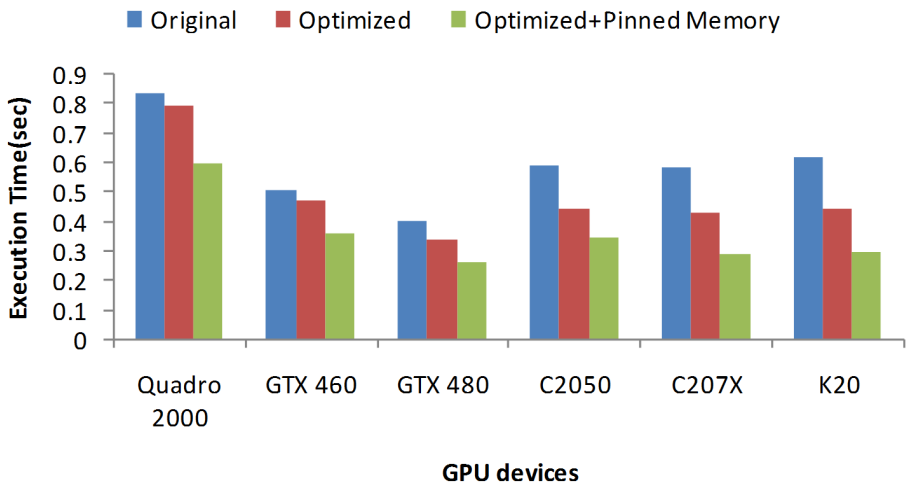
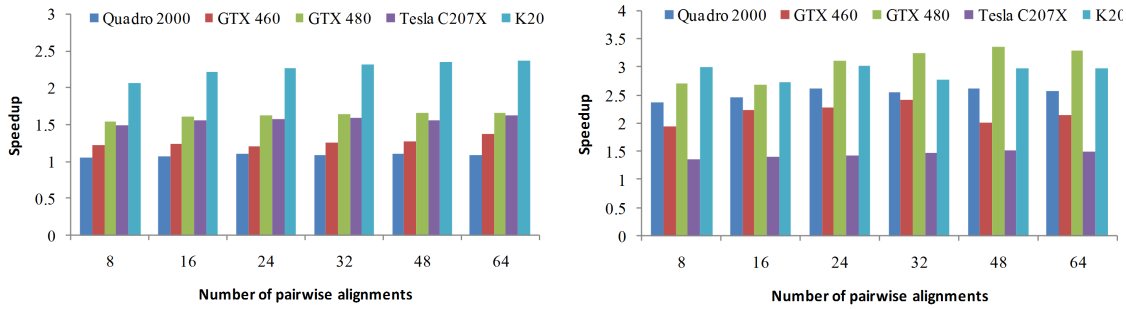


Figure 2.5: Evaluation of memory optimizations on Rodinia-NW: original implementation, optimized implementation with initialization of the alignment matrices on GPU, optimized implementation with pinned memory.

Memory Transfers: As explained in Section 2.2.3, Rodinia-NW initializes the alignment matrix on CPU and copies it to GPU. Also, to simplify memory access during computation, it creates a temporary substitution score table of size $m \times n$ during CPU initialization. For problems of the size considered, data transfer consumes considerable amount of time. An obvious optimization is to move the initialization from CPU to GPU. In addition, by omitting the creation of the temporary substitution table, more alignment matrices can be accommodated on the GPU, thus allowing for increased parallelism. In Figure 2.5 we show the effect of these optimizations

on different GPUs. In all experiments, 64 pairwise alignments are performed. The optimized version initializes the alignment matrices on GPU and avoids the initial CPU-to-GPU data transfer. On top of this, the optimized + pinned memory version uses pinned memory. As can be seen, the proposed memory optimizations lead to a 5-10% and a 20-25% decrease in execution time on low-end and high-end GPUs, respectively. In addition, the combination of the memory optimization with the use of pinned memory leads to a decrease in execution time in excess of 30% and 50% on low-end and high-end GPUs, respectively.



(a) TiledDScan-mNW (b) DScan-mNW

Figure 2.6: Kernel speedup over Rodinia-NW.

Kernel computation: We now focus on the performance of our compute kernels. Our analysis has two goals: (i) evaluating the performance improvements over Rodinia-NW, and (ii) devising criteria for selecting the optimal GPU implementation depending on the underlying GPU device. In Figure 8 we show the relative speedup in kernel computation time of DScan-mNW and TiledDScan-mNW over Rodinia-NW (the speedup is computed as the ratio between the compute time of Rodinia-NW and that of our GPU implementations). We performed experiments on all available GPUs and varied the number of pairwise comparisons performed from 8 to 64. Given its fine-grained alignment-to-core mapping, on these datasets RScan-mNW underutilizes the GPUs and reports poor performance. This, in general, holds when comparing long sequences on GPUs with 1-5GB device memory. Therefore, we focus on the other schemes.

Figure 2.6a reports the speedup of TiledDScan-mNW over Rodinia-NW. Note that TiledDScan-mNW performs fewer kernel calls (and therefore, has less kernel overhead) and involves more per-kernel computation (thus leading to increased parallelism). This motivates the performance improvement achieved by TiledDScan-mNW over Rodinia-NW. Note that the speedup increases with the computational power of the GPU (from 1.2x on the Quadro2000 to 2x on the K20). In fact, the increased parallelism in the TiledDScan-mNW kernel can be better serviced by GPUs with more SMs and compute cores.

As can be seen in Figure 2.6a, DScan-mNW also outperforms Rodinia-NW on all devices and datasets. Its performance is also generally better than that of TiledDScan-mNW, except on Tesla C207x cards. It is somewhat surprising that our approach does not show substantial speedup over Rodinia-NW on this device. It must be said that NW is an integer application, and Tesla GPUs are optimized for larger memory capacity (5GB vs. 1GB) and improved support for double precision floating point operations, but have a reduced clock rate (1.15GHz vs. 1.4GHz in GTX 480 cards, for example). We believe that the high number of uncoalesced memory accesses performed by DScan-mNW may motivate the poor performances on Tesla C207x cards, which have a slower memory clock.

Figure 2.7 reports the speedup of RScan-mNW over Rodinia-NW on sequences of different lengths. The number of pairwise comparisons performed in each experiment is reported on top of each bar (all experiments have been configured so to use 70% of the global memory capacity). As mentioned in Section 2.3, in RScan-mNW each core computes an alignment matrix: in order to fully utilize the computational resources of the GPU, RScan-mNW needs to perform a large number of parallel sequence alignments. This leads to pressure on the global memory capacity: for long sequences, the GPU global memory becomes the bottleneck, and the performance of RScan-mNW is penalized. RScan-mNWs performance improves when the length of the sequence

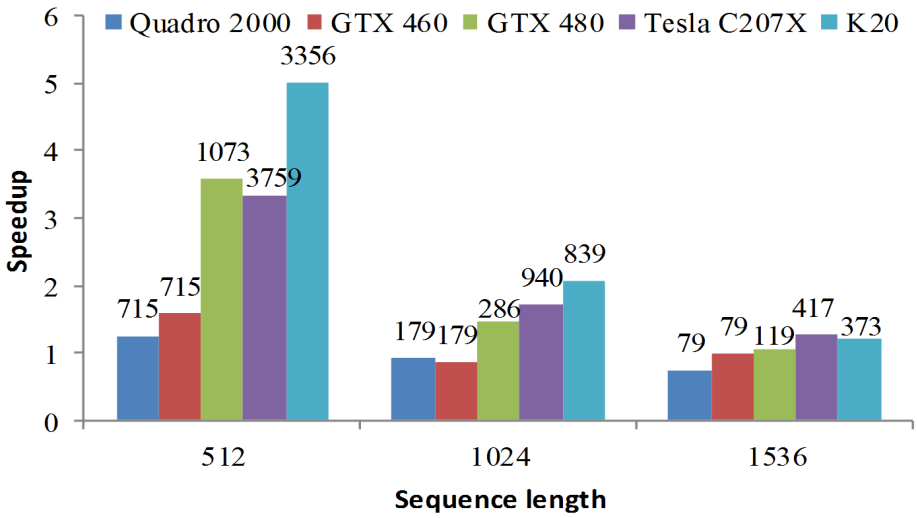


Figure 2.7: Kernel speedup of RScan-mNW over Rodinia-NW on sequences of various lengths. The numbers on the bars represent the number of pairwise alignments on the device for each experiment.

decreases: in the case of shorter sequences, the global memory can accommodate more alignment matrices, thus leading to higher utilization of the GPU cores. In particular, on 512-base sequences, RScan-mNW gives a speedup over Rodinia-NW up to a factor 5x.

In general, Figure 2.6 and 2.7 show that DScan-mNW and TiledDScan-mNW are preferable to RScan-mNW on the 1,536-base sequences in the 16S rDNA gene dataset. In addition, these results show that our methods overcome inefficiencies of Rodinia-NW, and suggest that DScan-mNW is preferable on all devices except Tesla C207x. On such cards, TiledDScan-mNW provides better performance. This finding will be used to configure our GPU-workers. As next step, we want to determine how to size the amount of work that each GPU-worker should pull from the GPU-dispatcher to operate at full capacity. In fact, we want to fully utilize the GPUs present in the system. The number of pairwise comparisons that can be performed concurrently on each GPU is limited by its memory capacity. We configured each GPU to operate with its global memory 75% full. For the sequence lengths being considered, this leads to 79, 79, 119, 208, 417, and 372 parallel alignments on Quadro2000, GTX460,

GTX480, Tesla C2050, Tesla C207x and K20 GPU, respectively.

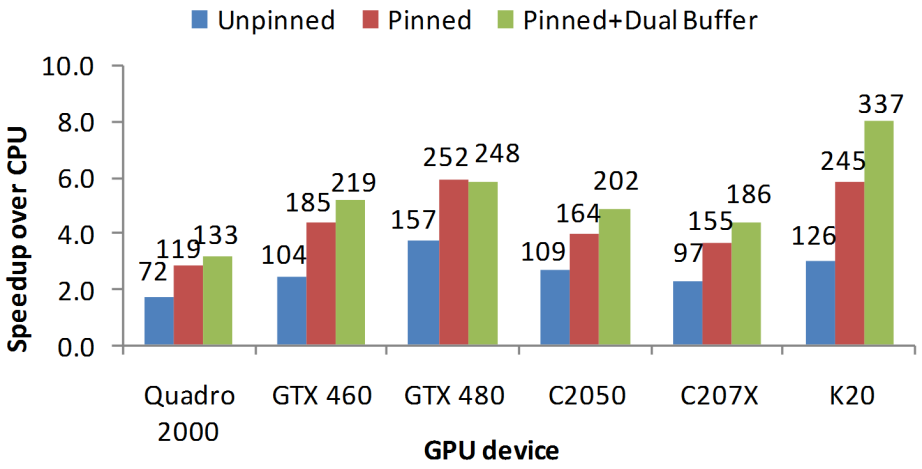


Figure 2.8: Speedup reported by DScan-mNW over an 8-threaded CPU implementation running on the 8-core CPU on Node-4. The numbers on the bars represent the processing throughput (number of pairwise alignments/sec).

Figure 2.8 shows the speedup reported by Dscan-mNW over an 8-threaded OpenMP implementation running on the 8-core CPU on Node-4 (2 x Intel Xeon E5620, 2.4 GHz, 48 GB RAM and 4 x GTX 480). The numbers on each bar represent the throughput in number of pairwise alignments/sec. For each GPU, we performed three experiments: one using unpinned memory, one using pinned memory, and one using dual-buffering. We first define an "optimal batch size" b_{SIZE} for a particular GPU to be the number of simultaneous alignments that can be performed given the device memory (as above). For the first two versions, we ran analyses consisting of a number of sequences equal to $3 \times b_{SIZE}$ in order to effectively time the computation. For dual-buffering, only half of the GPU memory performs alignments at one time, so 6 batches of size $b_{SIZE}/2$ were timed. The performance was measured as the number of sequence pairs compared per second.

As can be observed from Figure 2.8, switching to pinned memory offers a gain of roughly 1.6x, consistent with previous findings [115]. Not using the OS's virtual memory system could in principle limit the number of sequences that can be processed

Concurrent	Block size	Slice size (k)	Performance (k)
4	32	75	777
4	64	37	1943
4	**128**	**18**	**3987**
4	256	8	3691
4	512	3	1317
2	32	152	742
2	64	75	768
2	128	37	1935
2	**256**	**18**	**3924**
2	512	8	3527
1	32	306	315
1	64	152	740
1	128	75	768
1	256	37	1924
1	**512**	**18**	**3840**

Table 2.3: Performance of LazyRScan-mNW with different shared memory, block size, and slice size settings. Bold values represent the best results reported for each shared memory configuration.

concurrently. However, our observation is that problem sizes are instead generally limited by the amount of physical memory on the GPU, so we do not consider this CPU-based memory limitation to be a significant disadvantage. The application of dual-buffering along with pinned memory offers an additional average 1.2x speedup, with the exception of the GTX480 system, which does not show significant speedup. We speculate that the reason for this lack of improvement is that the GTX480 has a more restricted handling of CUDA streams, that does not allow the same level of overlapping of memory transfers and kernel computations possible on other devices. In general, it can be observed that even cheap low-end GPUs (like the GTX460 and GTX480) offer throughput in the order of 200-250 pairwise comparison/sec.

2.5.3 Performance analysis of LazyRScan-mNW on single GPU

As discussed in Section 2.3, thanks to its optimized global and shared memory usage, LazyRScan-mNW can potentially offer some speedup over the other three GPU kernels.

The critical parameter affecting the performance of this method is the slice size k, that is, the number of columns calculated by a thread before writing a column of the alignment matrix to global memory. Importantly, this parameter can influence the performance of the kernel in two conflicting ways. On one hand, larger values of k help threads to avoid expensive global memory read and write operations. On the other hand, smaller values of k reduce shared memory use so that the GPU can schedule more threads to run simultaneously, hiding the global memory latencies by alternating between warps waiting on memory and on computation operations.

We explored how the performance varies with k. In particular, we considered kernel configurations that fully use the global and shared memory available on the GPU. To this end, we set the batch size (that is, the number of alignments performed in a single kernel calls) so to utilize 80% of the global memory. We recall (Table 2.1) that LazyRScan-mNW stores in global memory l integer scores per alignment (l being the length of the longest sequence). Therefore, the batch size can be computed by dividing the amount of global memory used by $l \times 4$ bytes. In addition, in this method every thread performs a full alignment: thus, the kernel is invoked with a number of threads equal to the batch size. In our analysis, we started with thread-blocks of the size of a warp (32 threads), and performed several experiments by progressively doubling the thread-blocks' size up until 512 threads per block. Finally, we wanted to see if having multiple thread-blocks concurrently executing on one streaming multi-processor (SM) would improve performance. Thus, we varied the shared memory utilization between 25% (12KB), 50% (24KB), and 100% (48KB),

respectively resulting in 4, 2 and 1 thread-block concurrently scheduled on the same SM. The parameter k can then be calculated from the formula on shared memory utilization in Table 2.1: namely, by dividing the amount of shared memory per SM by the block-size times 5 bytes (that is, the amount of storage required for 1 integer and 1 char). To simplify this operation, we provide to potential users a spreadsheet along with our open-source code to help determine k for each of the scenarios above.

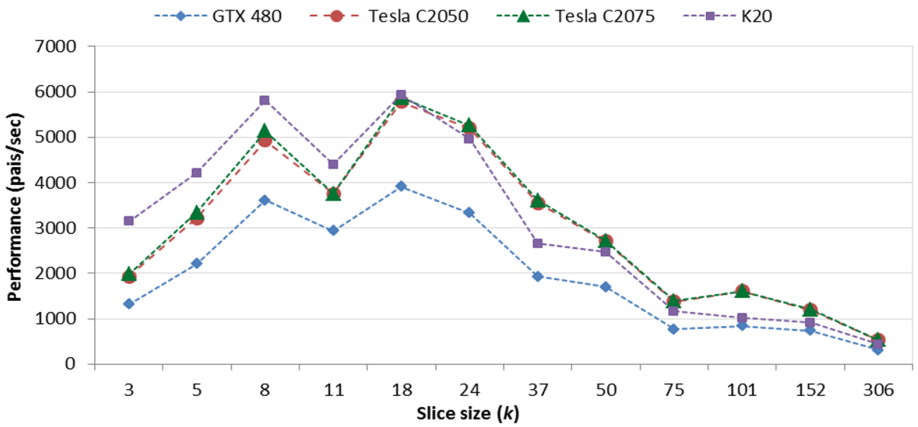


Figure 2.9: Effect of slice size k on performance of LazyRScan-mNW.

Table 2.3 shows the performance achieved on a GTX 480 GPU when varying the shared memory utilization, the block size, and the slice size k. As can be seen, the performance of LazyRScan-mNW is relatively consistent for fixed k when the shared memory and block size settings are changed. Specifically, the performance peaks at $k = 18$. When we repeated this same experiment on different GPU devices, we again found the same value of k to be optimal (Figure 2.9). These results can be explained as follows. In all configurations with $k = 18$, 512 threads are scheduled concurrently on each SM (that is, 4 blocks 128 threads or 2 blocks 256 threads or 1 block 512 threads). This suggests that scheduling 16 warps per SM is enough to hide the memory latencies, and, at the same time, a value of k=18 is large enough to reduce the amount of expensive global memory operations. As the characteristics of k remain relatively constant among different GPU devices, there seems to be some generality

Method	Action	Time (secs)	Overall (pairs/sec)
CPU Standalone	Alignment	2071.71	24.0
CPU+GPU DScan-mNW	Filtering	51.27	65.2
	Alignment	645.95	
CPU+GPU LazyRScan-mNW	Filtering	5.18	76.8
	Alignment	645.95	

Table 2.4: Speedup on a single node as a result of using DScan-mNW and LazyRScan-mNW as a preliminary filter for sequence analysis.

to these results. We also notice that the peak performances of LazyRScan-mNW (from about 4000 pairwise alignments/sec on low-end GPUs to about 6000 pairwise alignments/sec on high-end GPUs) are far better than those reported by the other three GPU kernels. Interestingly, the Kepler K20 GPU does not offer substantial performance improvements over Tesla C20XX devices. This can be explained by the fact that the memory latencies are in all cases sufficiently hidden by context switching between 16 warps. The K20 GPU, however, offers more favorable performance for small values of k. In this case, the more frequent global memory operations are better hidden by the massive multi-threading of the Kepler architecture.

2.5.4 Performance analysis on a single node, using GPU kernels as filtering methods

To evaluate our special case of computing all possible pairwise alignments but only retaining them if they meet a predefined percent identity threshold, we carried out performance tests on large-batch pairwise comparisons. These experiments differ from the ones presented in Section 2.5.3 in two ways. First, the dataset has a small number of sequences but leads to a large number of comparisons due to the large number of pairwise combinations. Second, after the alignment score is known, we need only perform the trace-back (on the CPU) to recover the actual alignment if a given sequence pair has an alignment score greater than the value of L above.

In the experiment, we created a FASTA file consisting of the first 200 sequences in our initial dataset (thus, making $200 \times 199/2 = 19900$ pairwise comparisons). The identity cutoff score is set to 97% (a common value, as explained in Chapter 5). Only sequence pairs that have identity 97% are selected for performing higher-quality trace-back and alignment on the CPU. In our dataset, the number of such pairs is 6095. As such, with an ideal filter that takes no time to do computation, the expected speedup assuming that all pairs take the same time to align is $19900/6095 = 3.26x$.

Table 2.4 compares the performance gains obtained by using either LazyRScan-mMW or one of our traditional NW implementations (dual-buffering pinned-memory DScan-mNW) as a "filter" over the original method of performing all pairwise comparisons on the CPU. To ensure the fairness of the comparison, we have eliminated the matrix transfer operations from DScan-mNW. The experiment was done on a workstation is equipped with a single low-end GPU card (GeForce GTX 460).

LazyRScan-mNW is the fastest implementation. As can be seen, both methods serve relatively well as a crude filter to avoid unnecessary alignments and offer a significant time saving even in the presence of a high-end CPU. However, the even greater performance improvement seen with LazyRScan-mNW is quite encouraging because it gives speedups that approach the maximum expected (3.26x, see above).

While LazyRScan-mNW is very applicable to our specific use case and provides significant speedup on a single node, it is also less flexible compared to the other three GPU methods, as the alignment is necessarily recomputed on the CPU (although we are currently studying mechanisms to efficiently perform trace-back on the GPU). Even with the trace-back not in place, the approach does still allow avoiding potentially expensive CPU operations or CPU/GPU communication and offers a considerable amount of time saving, proving to be a practical solution for important biological problems.

2.6 Conclusion

In this work, we have designed four implementations of multiple pairwise alignments using the Needleman-Wunsch algorithm on GPU. Three of our parallel kernels (TiledDScan-mNW, DScan-mNW and RScan-mNW) are general purpose. Our forth implementation (LazyRScan-mNW) is optimized for problems that require performing the trace-back operation only on a subset of the sequence pairs in the initial dataset (for example, the pairs which alignment score exceeds a predefined threshold). We have highlighted how the different computational patterns affect the utilization of the underlying hardware. We have integrated our general purpose GPU kernels with an MPI framework for deployment on homogeneous and heterogeneous CPU-GPU clusters. We have evaluated our framework on a real-world dataset and on a variety of low-end and high-end Nvidia GPUs. Our experiments based on the general purpose TiledDScan-mNW, DScan-mNW and RScan-mNW kernels show a throughput in the order of 250 and 330 pairwise alignments/sec on low- and high-end GPUs, respectively. In addition, we achieve a throughput of 1,015 pairwise alignments/sec on a 6-node commodity cluster equipped with a low-end GPU on each node. Our LazyRScan-mNW kernel allows throughputs up to about 4,000 and 6,000 pairwise alignments/sec on low- and high-end GPUs, respectively, and shows to be a very effective filtering method. Finally, we have performed an extensive experimental evaluation on the impact of the slice size on the performance of the LazyRScan-mNW method on a variety of GPU devices with distinct compute capabilities (2.0, 2.1 and 3.5). Our results can be used in a production setup to tune the code to the underlying hardware.

Chapter 3

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

This chapter has previously appeared in the substantially the same form as: Andrew Todd et al. "Parallel Gene Upstream Comparison via Multi-Level Hash Tables on GPU". in: *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*. IEEE. 2016, pp. 1049–1058. I contributed to this research in the Thread to Pair algorithm, k-mer encoding, hashing function implementation, design optimization, test data collection and filtering, and benchmarks.

3.1 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 [5], to genome assembly [107], to evolutionary genomics [110]. 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 [141]. 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" [35, 84]. 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 [82], 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 [35, 84], making them difficult to locate using a single genome [48]. Instead, researchers have used a comparative approach to find regions of DNA that appear to regulate the same gene in several related organisms [48, 60, 141]. 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 [98, 122]. Even more problematic, the regulatory motif bound by a transcription factor may change over time [154], making the comparison of genomes challenging. However, with the advent of new sequencing technologies [125], 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 [20], yeasts [16] and mammals [38] 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. The 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 transcriptor 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 [6, 58]. 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 [2, 3]. 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).

Our contributions are the following:

- We propose 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.

- We propose 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.

- We propose four GPU implementations of the all-to-all hash table comparison phase. Our GPU kernels leverage distinct parallelization approaches and differ in their computation and memory access patterns. We study how the selection of the hash function, the bucket size and other kernel-specific parameters affect the performance of our framework for motifs of various lengths.

3.2 Design of proposed motif-finding framework

High-Level Algorithm Design

Recall that the considered motif finding problem can be formalized as follows. Given n sequences of lengths $l_1, ..., l_n$, 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 that we use 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 naive 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 $\frac{\prod_{i=0}^{d-1}(k-i)}{d!}$ 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 $\frac{\prod_{i=0}^{d-1}(k-i)}{d!}$ insertions or counter updates in each Hamming 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 an high level representation of the data structures used in our implementation. Below, we provide details on our design.

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 $\frac{\prod_{i=0}^{d-1}(k-i)}{d!}$ 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 Section 3.3.

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.

Bucket Representation

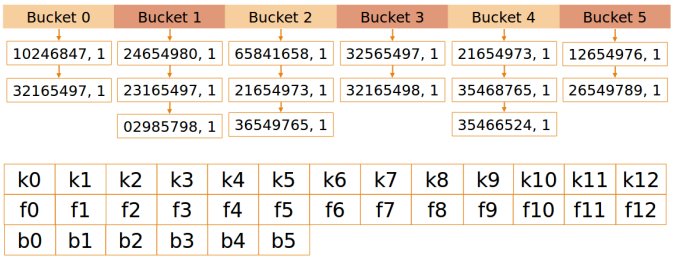


Figure 3.1: (Top) Pointer-based chaining hash table containing a set of k-mers. The unsigned tuple of values in the buckets indicate the 64-bit compressed representation of each k-mer, and an occurrence frequency counter. (Bottom) Compressed array-based representation of the same hash table, where ki represent the kmers, fi their occurrence frequency counters, and bi the offsets of the buckets within the arrays of k-mers and counters

For hash table construction on CPU we use a pointer-based bucket representation (Figure 3.1). 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

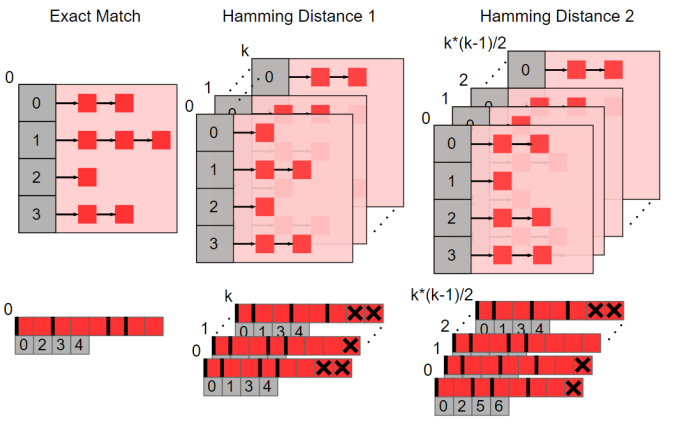


Figure 3.2: Hamming distance meta-tables and their respective sub-tables before and after array based conversion.

transferring the hash tables onto GPU for comparison, we convert all sub-tables into array form (Figure 3.1 and 3.2). 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 [116], this allows a reasonably optimized CPU performance to measure against.

Algorithmic Analysis Construction vs. Comparison

The number of k-mers that can be found in a sequence of length l is (l-k+1), so the corresponding hash table creation involves (l-k+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.

Hashing Methods

For hash table construction, we tested the following hash functions (their bit grouping schemes are shown 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:

```
1 hash(unsigned int kmer){
2     hash = 5381;
3     for all " 3 (or 2) bit groupings"{
4         hash = hash*33+"current bit grouping"
5     }
6     bucket_in_table = hash % buckets_per_table
7     return bucket_in_table;
8 }
```

Figure 3.3: DJB2 hashing algorithm.

In our 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 hashing contribution in the CUDPP library [2, 3], we used a 2-universal hash function of the form:

where k is the key, a and b generated by the Mersenne Twister PRNG, p is the

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

Figure 3.4: 2-U MT hashing algorithm.

largest 32/64-bit unsigned prime, and B is the number of buckets in a sub-table. In our 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 predefined 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 to 8-bits for 256 buckets).

This method operates as follows:

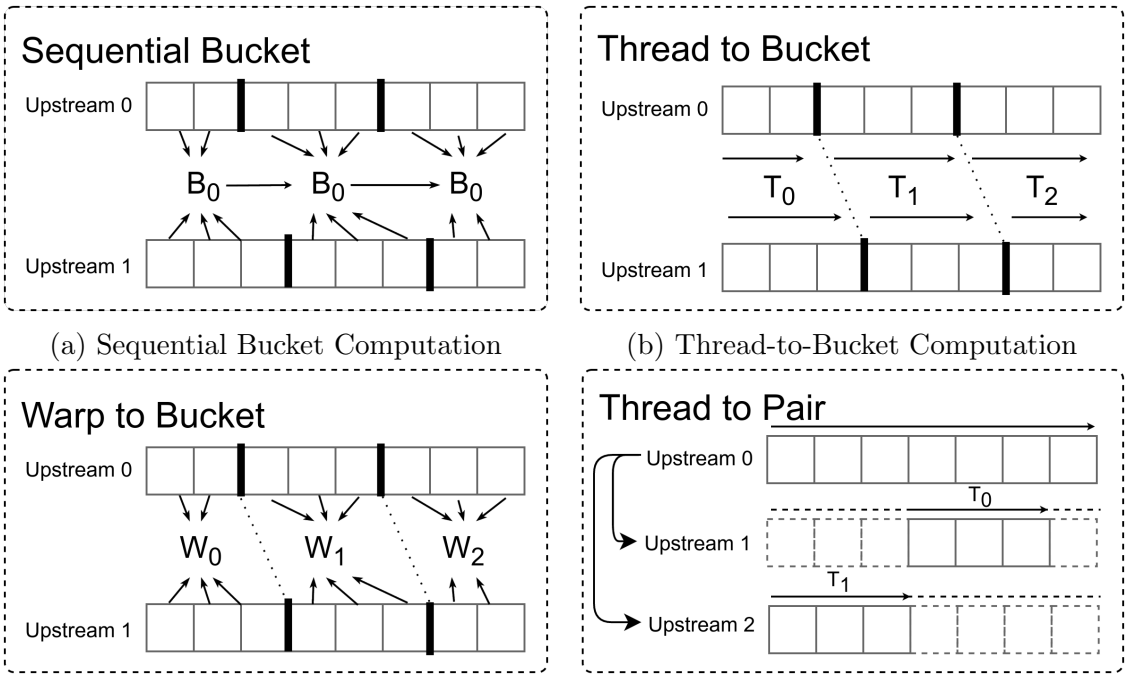
```
hash(unsigned int kmer){
    hash = unsigned int kmer;
    for all "r-bit groupings"{
        hash = hash XOR "current r-bit gourping"
    }
    bucket_in_table = hash % buckets_per_table;
    return bucket_in_table;
}
```

Figure 3.5: Custom XOR hashing algorithm.

where 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.

3.3 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



(a) Sequential Bucket Computation

(b) Thread-to-Bucket Computation

(c) Warp-to-Bucket Computation

(d) Thread-to-Pair Computation

Figure 3.6: 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, 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.

Background on GPU architecture and programming

Our GPU implementation is written using CUDA [93]. 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).

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 $\frac{n\times(n-1)}{2}$ pairwise comparisons that can be done in parallel. Each upstream region is associated d hash tables, resulting in $1 + \sum_{j=1}^{d} \frac{\prod_{i=0}^{j-1}(k-i)}{j!}$ 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 (as bucket comparing requires all buckets to be intact in order to be exhausted). Hence, using a large number of buckets during creation provides more flexibility in the comparison phase. Finally, as anticipated in Section 3.2, 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 3.1) to be stored in L1 cache by using the *__ldg* intrinsic.

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 3.6 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. 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 B_i and B_j is parallelized as follows. Each thread of a warp is assigned one or more k-mers from B_i in a contiguous fashion and compares such k-mers with all k-mers in B_j 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 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 is able to perform a massive number of pairwise hash table comparisons (as many as the number of threads launched) in parallel.

3.4 Related work

Automata-based approach

The motif-finding problem considered can be also addressed through a finite automata-based approach. Specifically, given an upstream region Ui of length l, the main idea is to construct the finite automaton that accepts all the k-mers found in Ui within Hamming distance d [123]. The k-mers that are common to Ui and any other upstream region Uj (or that differ in at most d characters) can then be found by traversing the constructed automaton using input Uj. 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 3.2), 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 [8], GPUs [17], and Microns Automata Processor (AP) [30, 114].

Logic-based FPGA designs [8] 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. [17] 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 APs 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) [114]. 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, Microns AP has a significant overhead associated with processing the output (match) information. This overhead could significantly affect the performance.

Other motif finding approaches

Most of the existing algorithms in this focus-area find possible motifs by using statistical methods to locate overrepresented k-mers [29]. 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.

Hash table implementations for GPU

Partly following the model of Lefebvre and Hoppe [65], we construct hash tables on the CPU and access them on the GPU. In contrast, Alcantara et al [3] pioneered a hash table approach performing both hash table construction and access directly on the GPU. The cuckoo hashing implementation of Alcantaras 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.

3.5 Results

Hardware and Software Setup

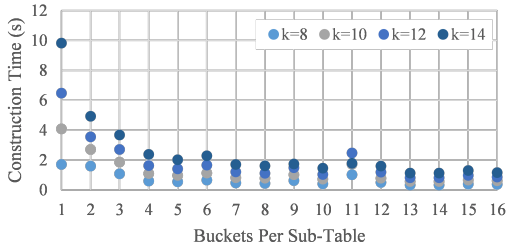
We run all experiments on an 8-core Intel Xeon CPU E-5620 v3 @ 2.40GHz with 64GB of RAM. The machine is equipped with an NVidia Titan X GPU with 12GB of RAM. The GPU is of Maxwell generation, has 24 streaming processors, and each streaming processor has 128 cores and 96KB of shared memory. The GPU runs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

Dataset

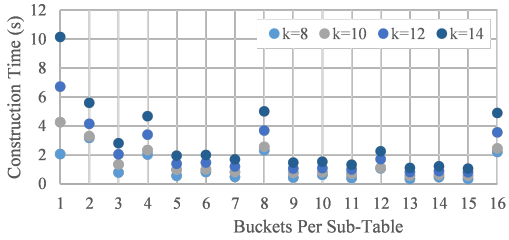
We extract the dataset from the Yeast Gene Order Browser [16] E. Cymbalariae genome. This genome has 4592 extractable genes and is average in terms of the number of genes among the yeast genomes. For each gene we consider both forward and reverse-strands of the upstream region, leading to 9,184 upstream regions. In

accordance with Harbison et al. [46], the extracted upstream regions are 500-base long.

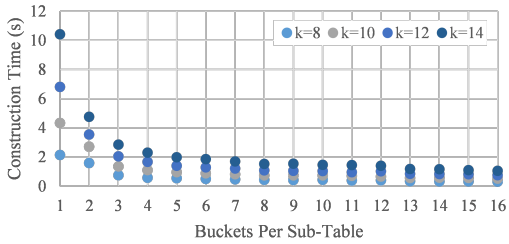
Hash Table Construction Performance



(a) DJB2-based construction



(b) 2U-MT-based construction



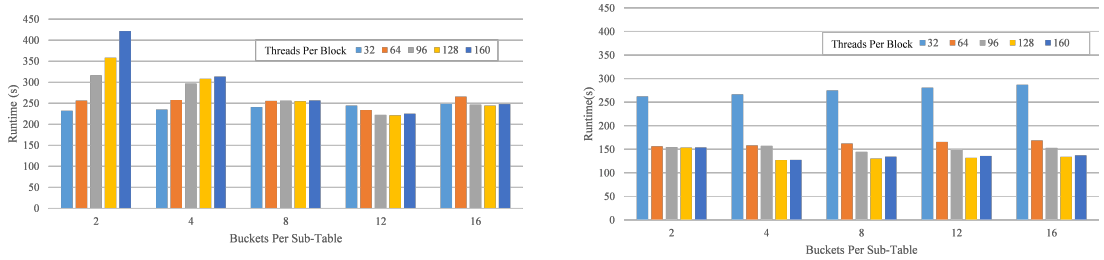
(c) Custom XOR-based construction

Figure 3.7: Construction time for three hash functions across different bucket configurations and k-mer sizes. Spikes in construction time correspond to hash unfairness.

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 3.7 reports the behavior of three approaches to hashing, 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 case, 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 decreases 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.

GPU Hash Table Comparison Performance



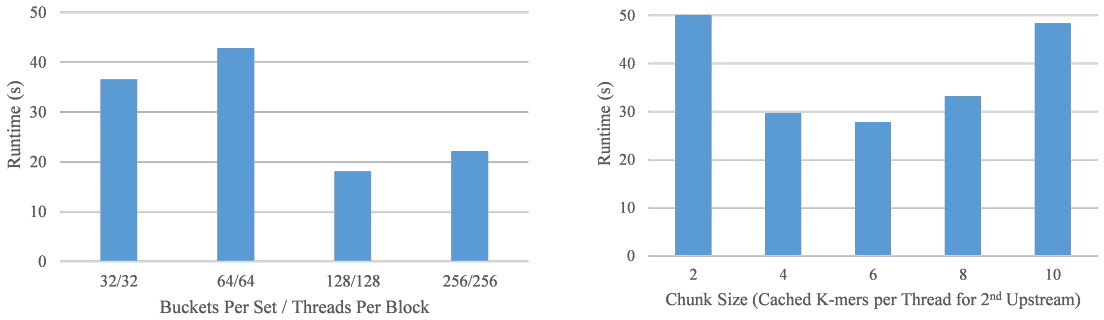
(a) Sequential Bucket kernel

(b) Warp to Bucket kernel

Figure 3.8: Execution time of the Sequential Bucket kernel and of the Warp to Bucket kernel for different bucket settings and block sizes.

Figures 3.8 and 3.9 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 Sequential Bucket implementation (left side of Figure 3.8) we note for



(a) Thread to Bucket kernel

(b) Thread to Pair kernel

Figure 3.9: Execution time of the Thread to Bucket kernel with both upstreams fully cached; 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. Both versions use 1024 blocks with k = 12.

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 3.8) 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 3.9), 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 3.9), 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 3.9) 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.

Final Speedup of All Comparison Phase Versions

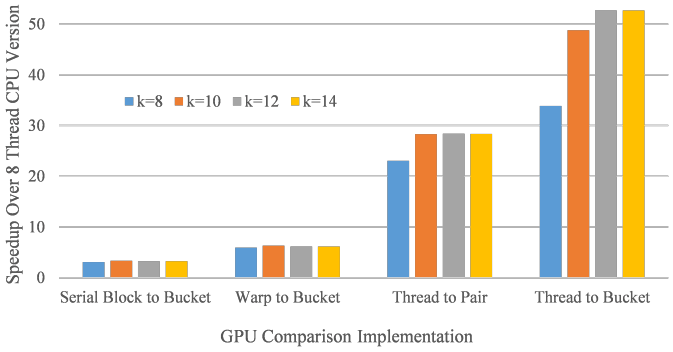


Figure 3.10: Speedup over 8-thread CPU implementation for different k and GPU implementations. Upstream Comparison is done such that both CPU baseline and optimized GPU versions work on the same array-based data-structure.

In Figure 3.10, 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.

3.6 Conclusion & 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 (with larger upstreams) 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.

Acknowledgment

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

Chapter 4

PolyHomology: A Comparative Genomics Approach to Regulation Pattern Discovery

4.1 Introduction

Eukaryotic genomes are notoriously complex and can contain tens to hundreds of thousands of genes [149][43]. Even for a well-studied model organism such as the budding yeast (*Sacchromyces cerevisiae*), where the number of genes is estimated to be only about 5,500, 28% the protein-coding genes are still unannotated according to PANTHER version 10 [86][16]. Understanding when and where these genes are expressed can give insights into what biological processes or molecular pathways they act in. The rules governing the expression of a gene are complex and incompletely understood. However, special proteins called transcription factors (TFs) play a critical role in binding to the DNA of a gene and enhancing or suppressing its expression (e.g., transcription [148]). This process is driven by the specific binding of those TFs to short sequences of DNA known as motifs. While genes can be identified with relatively high confidence across species, regulatory motifs identification tends to be

much more challenging. Even with a relatively good algorithm, the rates of false negatives and false positives are excessive [40]. A major reason for this difficulty is that the motifs are both short and degenerate (e.g., some positions in the motif can vary without altering binding), making it difficult to generate statistical models that detect them with high confidence [144]. Thus, the big picture of understanding what sets of genes are co-regulated is still largely unsolved, especially when one looks beyond model organisms to the ever growing list of sequenced genomes [39] [1]. At the same time, the problem of understanding gene expression would be considerably simplified if we knew, for each gene, all of the other genes with which it shared a common TF.

One promising avenue that has been opened by the advent of inexpensive sequencing is the use of the comparative method, one of the historically most successful approaches to biology [41] [32] [37] [157]. In particular, we hypothesize that very large-scale comparative genomics approaches, using dozens to hundreds of genomes from distinct species, might be able to identify shared gene regulation on a global scale using even fairly simply homology measures. In this model, we assume that important regulatory pathways (e.g., which TFs bind to which genes) have been conserved over evolutionary time [99]. Indeed, there is evidence suggesting that evolutionary pressures to maintain important biological functions and pathways lead to corresponding orthologs being conserved over time [140] [95]. Our approach assumes that both the binding motifs and their recognition elements in the TF proteins may drift over time, but that selection acts to maintain the binding of the TF to its target genes. Given this assumption, it follows that shared similarity in upstream regions between co-regulated genes, if also present, albeit with different motifs, in orthologous genes in other taxa, is evidence of co-regulation. Coupled with rapidly falling costs for DNA sequencing, we propose that it may be possible to identify co-regulated genes through the comparison of all upstream regions against each other in each genome and then

integrating these potential regulatory sites across a large number of related genomes using the principle of orthology among the genes analyzed. From this premise, we here propose PolyHomology, a method that combines information from homologous genes across species, to find such highly co-regulated genes using a big-data approach. In a sense, this approach is a simplification of the phylogenetic footprinting technique [21, 14]. However, we differ with traditional footprinting technique in two significant ways: First, we forego the estimation of the regulatory motifs themselves in an effort to improve the prediction of co-regulated genes. Second, we do not require a target gene of interest. Hence, while some of the yeast genomes analyzed here were previously used for a footprinting analysis [59], that analyses focused only on closely related species where motif conservation was expected, whereas we use a much more diverged set of genomes. As such, the goal of this work is not the prediction of co-regulated genes per se but rather assessing if such a prediction would be possible in a future world with extremely densely sampled genomes. Because analyses based on such dense sampling presents formidable computational challenges, we developed our algorithms to exploit the parallelism inherent in these data by using GPU co-processors for the most computationally demanding part of the analyses.

4.2 Materials and Methods

In this section, we provide an overview of the method used to compute the pairwise k-mer composition similarity score between gene upstream regions in each genome. Then, we describe our algorithm to construct local- (one genome-) and meta- (multiple genomes-) graphs of co-regulation based on these pairwise upstream similarity scores. We employ a baseline metric of well-known co-regulated genes, to evaluate the effectiveness of the method.

4.2.1 Materials

Our hypothesis proposes comparing the regulatory regions of large numbers of orthologous genes to infer shared regulation. We are fortunate to have a curated set of yeast genomes with exactly this orthology information already available, specifically from the Yeast Gene Order Browser (YGOB) project, database version 7 [16]. There are 20 genomes represented in YGOB, split between 12 species possessing the yeast whole genome duplication (WGD) event [161] and eight lacking this WGD. From YGOB we obtained both the genome sequences themselves (in FASTA format) and a set of 14101 *pillars*, with each *pillar* corresponding to a set of syntenically orthologous genes.

To extract the upstream regions (URs) of genes, we developed an in-house Go script named UpstreamX using the BioGo library [63]. UpstreamX reads from the whole genome and outputs the extracted upstream regions for each gene, with up to N basepairs. Here, we decided to follow standard practice of extracting both in both Watson and Crick strands and without overlaps [148]. According to Harbison et al. [47], the vast majority of transcriptional regulator binding sites are within the 500 base pair of the transcription start site. Thus, we selected $N = 500$ for this analysis.

4.2.2 K-mer composition similarity score

We chose to use the presence of shared k-mers as our signal of shared regulation in upstream regions. However, because of the degeneracy of regulatory regions, considering only exact k-mer matches in upstream regions is likely to fail. Hence, we required a program able to compute near matches of $distance - n$ within k, where $n < k$, so that similar (but not necessarily identical) upstream regions that are co-regulated by the same TFs are accounted for: high similarity scores in such cases are desired, as they reflect many shared k-mers between a pair of genes.

We developed a GPU-accelerated program, SequenceMatch [143], to count the shared *k-mers*, i.e., subsequences of length k between pairs of URs in a single genome as produced by the UpstreamX program. The parameter k is configurable from 6 to 30 nucleotides for each analysis. Specifically, given each upstream i has n_{Ki} occurrences of *k-mer* K, then the value for all the *k-mer* matches between upstream i and j is the minimum of occurrences of K in i and j:

$$exact(i,j) = \sum_{all\ K} min(n_{Ki}, n_{Kj}) \tag{4.1}$$

To account for near-matches, we also used the same method for counting $distance-1$ matches and $distance-2$ matches in the computation. We achieved 52x acceleration running this comparison algorithm using a commodity NVIDIA GPU (GTX 960 with 4GB RAM) compared to a similarly optimized server-grade Intel Xeon CPU (4-cores E5-1607). We next describe how we computed an overall score that captures the *exact*, $distance-1$, $distance-2$ scores of every pair of upstream regions.

For simplicity, we assumed that all bases are equally likely to occur in all upstream regions. Thus, an *exact* match automatically leads to k $distance-1$ matches and k^2 $distance-2$ matches (of course the existence of $distance-1$ matches does not similarly imply the existence of an *exact* match). We define the *weight* w as the relative importance given to an *exact* match compared to a $distance-1$ match, as well as to a $distance-1$ match compared to a $distance-2$ match. Thus a natural value for w is k. Internally SequenceMatch computes *exact* matches by creating one hash table for each upstream sequence. By the same account, the program computes $distance-1$ and $distance-2$ matches by creating k and $k*(k-1)$ hash tables, respectively. Therefore, the memory requirements and requirement of each additional mismatch level is in the order of $O(k)$. The overall matching score is then computed in Equation 4.2 as follow:

$$score(i,j) = distance - 2(i,j) + w \times distance - 1(i,j) + w^2 \times exact(i,j) \quad (4.2)$$

After obtaining the scores for every pair of URs for each genome, a local graph for that genome is constructed. In this local graph, each UR (and by extension each gene) is represented by a node; pairs of nodes are connected by edges having a weight representing their overall matching score.

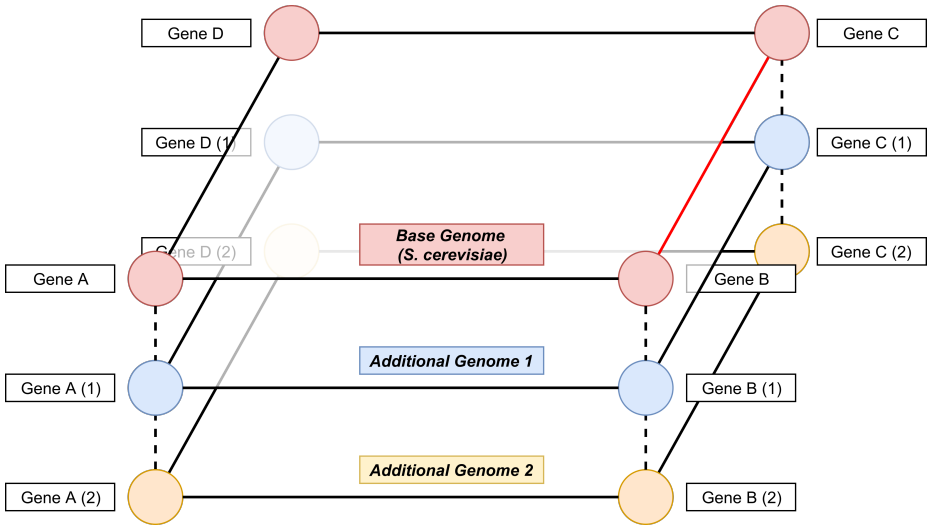


Figure 4.1: Local- and meta graph construction. A model of the meta-graph construction. Each plane is a local graph constructed for a given genome, depicted by different node colors. In each of the genomes, each node depicts a gene. Edge weights between any pair of nodes are determined by the number of shared *k-mers* between the corresponding URs in that genome. The meta graph is then constructed by stacking orthologous genes on top of each other. In the base genome *S. cerevisiae*, an edge is marked red if the two corresponding genes share at least one common transcriptional factor, known from Harbison et al. [47], which serves as the baseline sensitivity measurement.

4.2.3 Meta graph construction

We next constructed a meta-graph by merging the local graphs from a number of genomes using the orthology information from YGOB. The WGD event in some of

these yeasts adds an additional complication: we thus chose to only analyze post-WGD genomes. Ohnologs (paralogs arising from the WGD event) are thus distinct and treated as separate orthologous genes. Orthologs are anchored to corresponding *S. cerevisiae* gene, then merged as one gene in the meta graph. In the local graphs of species other than *S. cerevisiae*, genes that do not have corresponding orthologs in *S. cerevisiae* are discarded.

After all points in each genome (each layer of the graph) are anchored, the local graphs then can be overlaid one to another. Edge weights are calculated as the arithmetic mean of the edge weights in the local graphs (Fig 4.1). Note that when adding additional genomes, we do not require that every node be present in every genome, however we consider *S. cerevisiae* genes that have less than an arbitrary number of orthologs in other genomes to be a weak signal. Depending on the specific analysis, the weak data points can be discarded (See Section 4).

4.2.4 Sensitivity evaluation

From the Harbison et al. study [47], we extracted a set of potential co-regulated genes in *S. cerevisiae*. In this dataset, the DNA-binding profiles of 203 TFs were assessed across different environmental conditions. These inferred TF-binding propensities were mapped to their corresponding genes in *S. cerevisiae*. The resulting dataset consists of a matrix of $p - values$ giving a measurement of the likelihood of each TF binding to the upstream region of every gene in the genome. We identify the potential co-regulated set as all pairs of genes that are regulated by at least one common transcription factor (selected by $p - value < 0.001$). The edge connecting any upstream pair of potentially coregulated genes is then marked red in the graph.

4.3 Results

Using 12 yeast genomes, we explored our hypothesis that increasing volumes of genomic data might allow researchers to uncover regulatory relationships with simple sequence comparisons. For all pairs of genes across all genomes, we computed a statistic s, measuring the number of shared short sequence motifs (k-mers) shared in their upstream regions (URs). We then explored the ability of this network approach to identify gene pairs known from experimental data [47] to share regulation (e.g., they are targets of a common transcription factor.)

4.3.1 Initial graph construction and initial parameter estimation

The first step of our analysis was to construct a local graph from the URs of *S. cerevisiae*, with nodes representing genes, and edges representing their relative similarity in k-mer composition. The k-mer composition similarity is described in Equation 4.2. To construct the meta graph, we added further genomes in order of increasing phylogenetic distance, using the set of species relationships from the phylogeny presented by Gordon et al.[42], [24].

We explored how the structure of the meta graph varied with different values of k, the length of the *k-mer* being considered. In particular, we asked how the value of k affected the sensitivity in the detection of known co-regulation patterns. Recalling that motif lengths range from 6 to 12 nucleotides, we set $k_{min} = 6$ and $k_{max} = 14$, a choice motivated by the fact that the software can efficiently accommodates motifs with up to 14 identical nucleotides along with 2 unmatched ones [143]. Additionally, for each value of k, we used values of w ranging from k_{min} to k_{max}.

Figure 4.2 shows how the sensitivity of co-regulation detection varies with k. We observed that with only the *S. cerevisiae* genome in the meta graph (the meta graph

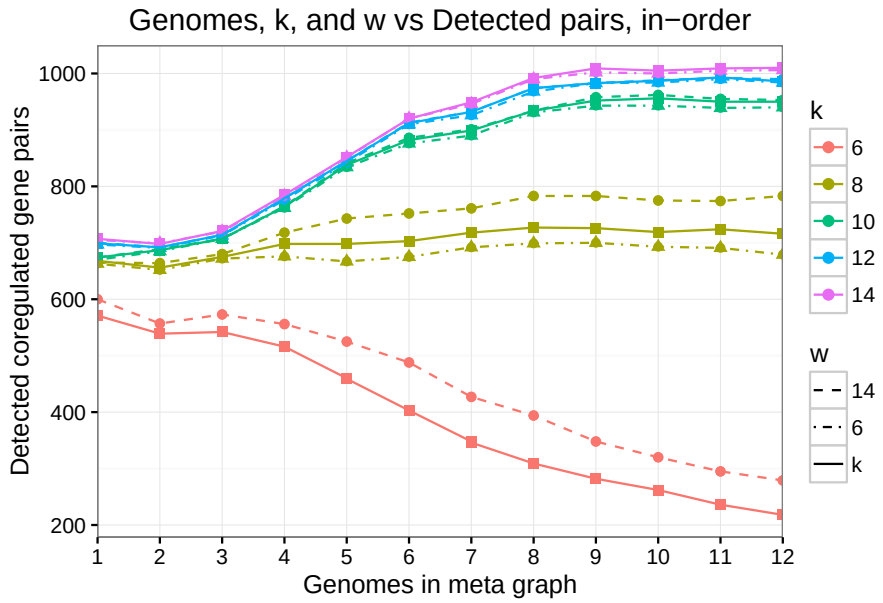


Figure 4.2: Sensitivity of co-regulation detection with variable k. Genomes are added in phylogenetic order from the single, reference *S. cerevisiae* genome up to the inclusion all 12 post-WGD genomes. Solid lines denotes the default parameter of $w = k$. We also tested values of $w = 6$ and $w = 14$, identified by dot-dash and dashed lines, respectively. k seems to play a more important role than w in determining overall sensitivity. With lower values of k, adding more genomes to the meta graph even reduces sensitivity in some cases. With higher values of k, having more genomes improves sensitivity, however different values of w do not yield significantly better or worse results.

that is identical to the local one), the choice of k does not strongly influence the sensitivity. However, as more genomes are added to the meta graph, the behavior diverges depending on the value of k. When $k < 10$, performance either remains approximately constant as genomes are added ($k = 8$) or actually degrades ($k = 6$). On the other hand, with $k \geq 10$, there is effectively a steady increase in the sensitivity as genomes are added. For $k = 10$, sensitivity effectively plateaus before all twelve genomes have been added, and the sensitivity is relatively invariant with respect to the value of the weight parameter w. Both $k = 12$ and $k = 14$ offer slightly better sensitivity than does $k = 10$. However, we do not believe the greatly increased computation time and memory needed for analyses where $k > 14$ are likely to be justified, given the small magnitude of these improvements.

4.3.2 Network and graph composition

We constructed these graphs using an initial parameter $k = 8$ and $w = k$, representing our best prior estimate of appropriate values. Thus, the parameter $k = 8$ was chosen because motifs generally have lengths between 6 to 12 nucleotides [72], [105], [127].

The annotation of the *S. cerevisiae* genome used had 5988 genes, resulting in 17,925,078 pairwise edges. For the purposes of visualization, we considered only the 5,000 edges with highest scores. Among the selected pairs, we then counted and marked the number of pairs of genes that are likely co-regulated by defining them as having at least one TF in common. Pairs that share more than 1 TF are treated no differently. In the graph, the edges connecting such pairs of genes are marked red. The meta graph with 12 genomes is then visualized with Gephi and then clustered with the Yifan Hu Proportional algorithm [54]. Figure 4.3 shows part of the graph (It is unhelpful to show the full graph as it is fully connected by definition).

4.3.3 Sensitivity to other factors

We further investigated a wider range of w to see if significantly larger or smaller values of w ($1 \leq w \leq 100$) can affect our results, using a fixed k=10, which has acceptable computational performance, with $O(k^2)$ hash tables for the algorithm employed in *SequenceMatch* [143]. In addition to evaluating values of w as given in Equation 4.2, we also experimented with the same set of values of w, but with the $distance - 2$ scores discarded. If discarding $distance - 2$ scores does not significantly affect our result, then the analysis time and memory requirements can be drastically reduced (from $O(k^2)$ to $O(k)$) because the $distance - 2$ hash tables are not computed.

Again, with an appropriate value of k, even quite different values of w did not negatively affect the sensitivity. Moreover, discarding the $distance - 2$ matches also did not reduce the detection sensitivity appreciably.

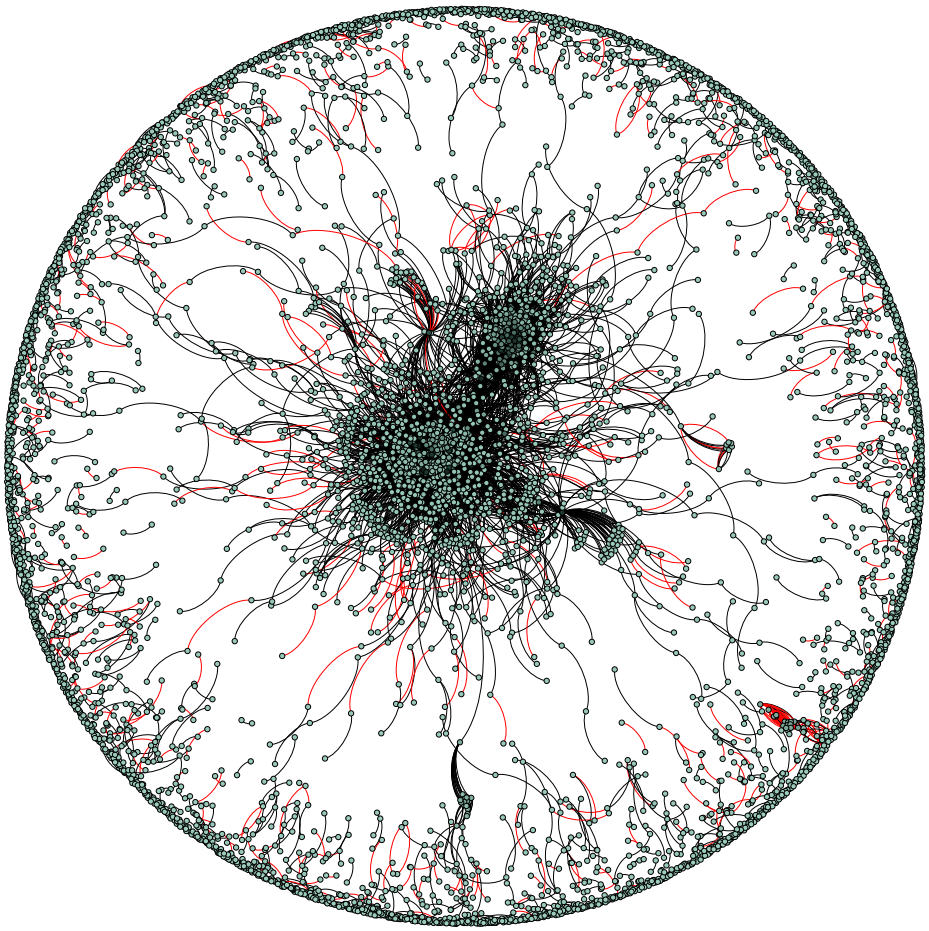


Figure 4.3: Initial meta graph with parameters $k = w = 8$. For clarity, only 5000 edges with the highest weights are shown. In addition, only non-orphan nodes are shown (e.g., genes present in at least one additional genome besides *S. cerevisiae*). Nodes with higher intensity of blue are those higher degree. Red edges indicates co-regulated edges according to data from Harbison et al.,. The cluster in the center centers around SCR1, snR53, YOR92C, snR33, YGL051W, tRNA genes, and others. Visually, it is possible to identify several large clusters in the meta graph.

4.3.4 Sensitivity with different orderings of genome addition

Next, we experimented with different orders with which genomes were added to the meta graph. The question we wanted to address was whether having more genomes always improves the sensitivity, even if the added information is from distantly-related genomes. With each iteration, we shuffled the ordering of the 12 genomes used and added them in that random order. We used the parameter of $k = w = 10$. When all 12 genomes are added to the graph, the result converges to the same number of edges

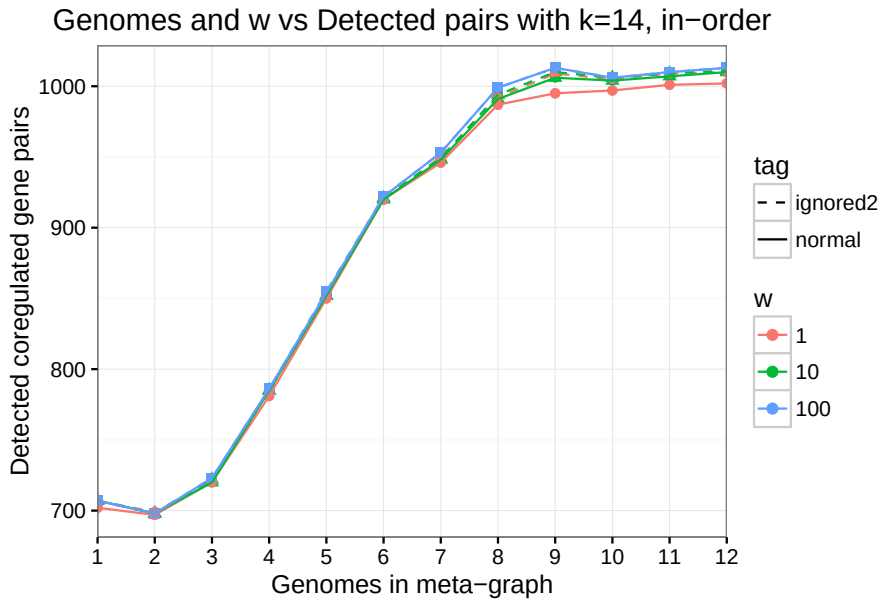


Figure 4.4: Sensitivity of co-regulation detection to differing values of the mismatch weight parameter w. With a well-selected value of k, the sensitivity usually increases as more genomes are added, regardless of w. Additionally, omitting the $distance - 2$ matching when computing the pairwise score does not generally degrade sensitivity.

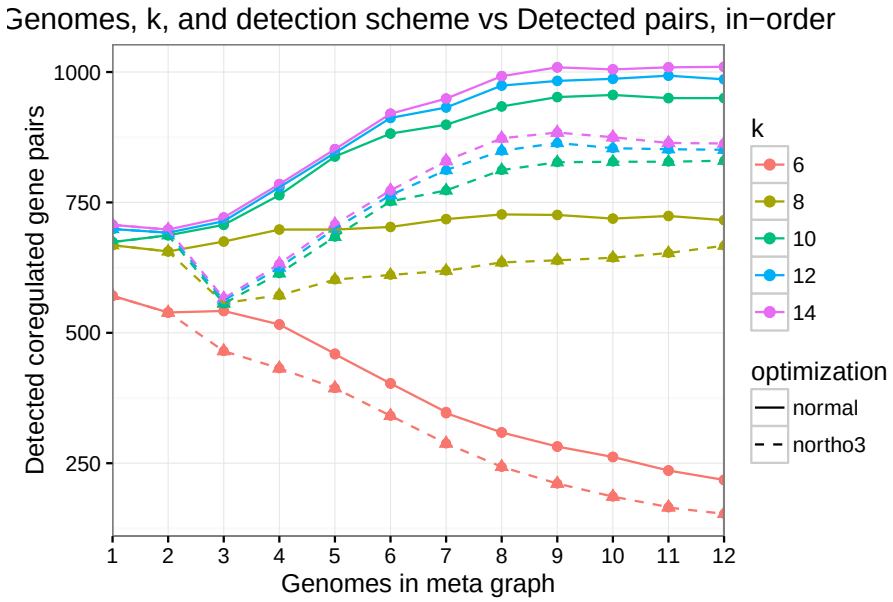


Figure 4.5: Sensitivity of co-regulation detection with differing values of the mismatch weight parameter w. With a well-selected value of k, the sensitivity usually increases as more genomes are added, regardless of w. Additionally, omitting the $distance - 2$ matching when computing the pairwise score does not generally degrade sensitivity.

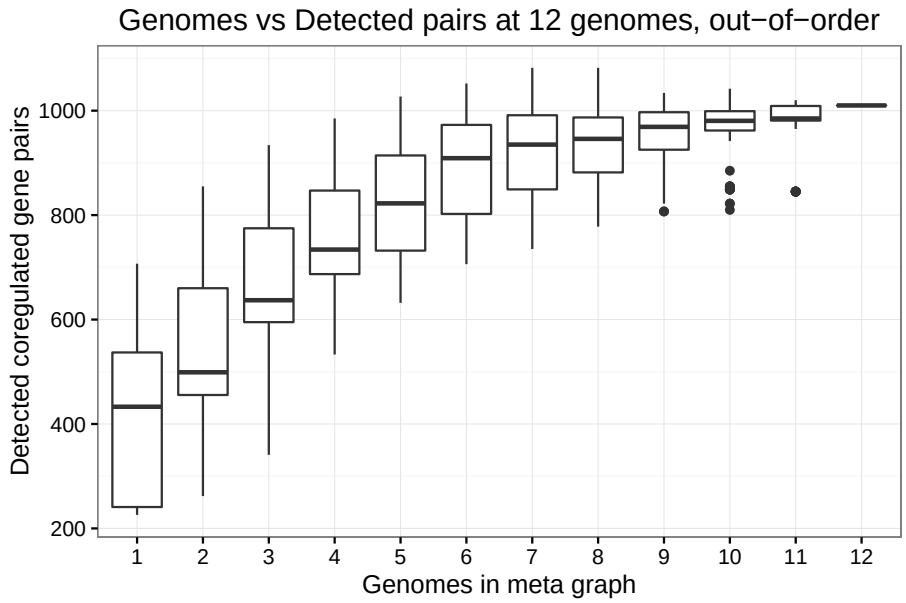


Figure 4.6: Sensitivity as more genomes are added in random order, 100 replications. Rather than a phylogenic order of genome addition, we randomly selected genomes to add to assess how performance changes when phylogenetically distant genomes are added early in the inference process.

detected due to the averaging nature of the method. We observed that the sensitivity does indeed increases as more genomes are added, regardless of order.

4.3.5 Sensitivity with Hughes pairwise data

The binding of transcription factors to DNA is only one aspect of a complex process of gene expression. To explore more how our proposed approach is able to detect real co-expression relationships "on-the-ground," we created a coexpression network from the large expression compendium of Hughes et al. [56].

The coexpression network we inferred from the Hughes data [56] consisted of 146,360 pairs of genes with a Pearson's correlation coefficient greater or equal to 0.5. We filtered these pairs and retained only those with correlation coefficients greater or equal to 0.75, resulting in 10,831 pairs of coexpressed genes. We then ran the same protocol with varying k, w ranging from 6 to 16 and asked how many true positives

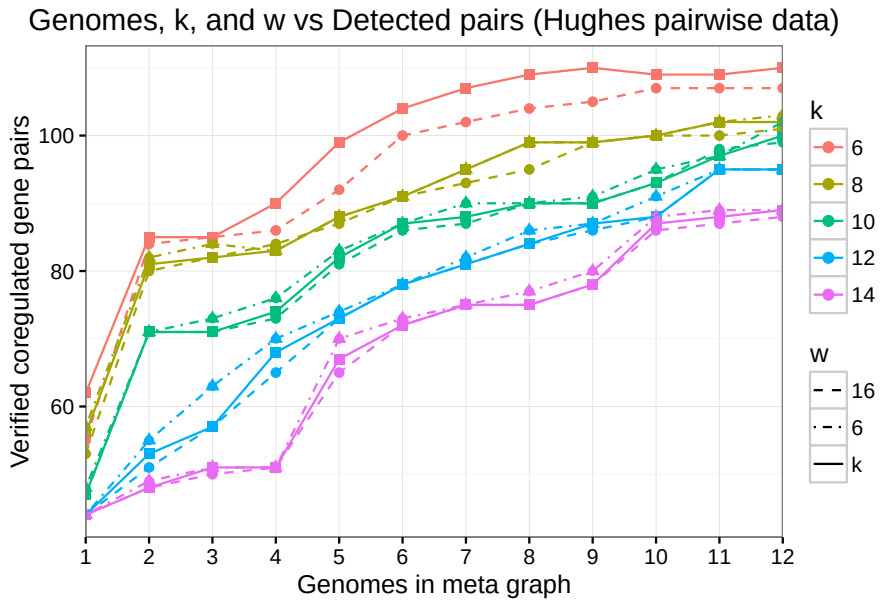


Figure 4.7: Sensitivity as more genomes are added in order. The Y axis indicates the number of co-expressed pairs appearing on the Hughes dataset among the top 1000 pairs that PolyHomology predicted.

(e.g., edges in the meta-graph that are also edges in the coexpression number) are there in the top 1,000 results that PolyHomology predicted. Figure 4.7 shows the number of verified gene pairs that have strong correlation coefficients in the Hughes data.

Again, the sensitivity increases as more genomes are added. However, it is notable that all parameters of k yield monotonically better sensitivity, in contrast to the previous results from the TF binding data, where only larger values of k yield better results.

4.4 Discussion

4.4.1 Overall effectiveness

In general, the use of comparative data via our layering method improves the sensitivity to the detection of co-regulation. The method is sensitive to the k-mer parameter

selection. However, contrary to our expectation, the weight of the $distance - 2$ matches does not seem to meaningfully contribute the overall sensitiveness. This result suggests that it is possible to simplify and computationally speed up the analysis significantly.

The k parameter selection in k-mer composition similarity scoring tends to prefer larger k values, we believe this behavior is due to the fact that chance matches should be much rarer for larger k values: When $k = 1$, for example, then every UR is expected to match every other UR with an identical score. The k parameter that yields better sensitivity, however, might not represent the most typical motif lengths, and should not be thought of as such. For instance, a motif of length 10 consisting of 4 fixed nucleotides, followed by 2 variable nucleotides then followed by 2 fixed nucleotides is accounted just as well by $k = 4$ as $k = 10$ with $distance - 2$. The fact that higher k does better in sensitivity might suggest that shorter k values result in spurious random $k - mer$ matches.

4.4.2 Limitations and future improvements

Manual inspection reveals that the average of the scores by adding additional genomes in many cases do not increase the amount of signal: Many of the top-scoring edges in the meta graph remain to be single data points (edges with no orthologs in other genomes). We experimented with artificially lowering the scores of such edges, but it does not seem to resolve the problem.

We noticed among the top 100 edges with the highest edge weights of the list of detected co-regulation patterns at $k = 10$, the algorithm confirmed 9 pairs co-regulated by YAP5, 1 pair co-regulated by YAP6, and 1 pair co-regulated by MSN4.

Currently, SequenceMatch only performs simple counting of scores with the assumption of equal probability of bases in the sequences. These assumptions conserve memory and simplify computational resource requirements .However, the program

might not be able to accurately account for GC-content and might slightly skew the results. We plan to incorporate this additional feature in future research.

As noted, we do not argue that the data presented currently represent a useful means for identifying coregulated genes. Rather, we argue that as more genomes are sequenced, the patterns of shared upstream k-mers may be useful data in the detection of such coregulation. We have not sought to estimate the number of genomes needed for such inferences because the true positive datasets we used here are likely incomplete. Moreover, many existing motif inference methods are computationally demanding – recent pipelines expect such algorithms to run on clusters with hundreds of nodes, such as in the case of DMINDA [79]. PolyHomology required around only two days of commodity GPU time to compute each local graph used here even with distance 2 matches. As the approach seems to also work even when distance 2 matches are not considered, the approach can be even ported to a CPU-only solution. Hence, we estimate that a sample of 100-200 genomes could be compared in the order of hours. This fact shows that our approach scales to the problem sizes relevant for an era of ubiquitous genomics and citizen participation in science and discovery.

Chapter 5

Operational Taxonomic Units classification: Diving into Phenetics Approach with the 16S rDNA Subunit

5.1 Introduction

Metagenomics represents an integration of genomic technologies with questions and theories from ecology and microbiology. While genomics is focused on studying species, meaning that it must be possible to obtain DNA from a single species (or indeed most commonly a single individual), metagenomics' focus is on studying the composition of all the microorganisms living in a specific environment. Thus, metagenomics can give answers in cases where classical genomics cannot, for example, when the species of interest does not grow well or in isolation in the lab environment [145]. More importantly, the quantification of abundance, composition and interactions of the species are crucial to understanding and comparing environments [145] and in linking surveys of microbial diversity to broader ecological theories [83]. Metage-

nomics also has linkages to other areas of biology–in particular to understanding the various interactions (mutualistic, parasitic and commensalistic) between microbes and multicellular organisms, especially interactions with animals [67] and plants [15]. A striking example of the utility of metagenomics was the observation that an obesity phenotype can be transferred to nonobese animals through changes in their microbiomes [147]: other examples focusing on the information contained in the taxonomic profiles of metagenomics include differences in the microbiomes of forage-fed sheep verses concentrate-fed sheep [34] and diseased verses normal hosts [111], [75].

Analyses of microbial communities such as these that are based on taxonomic classification of the individuals found in an environment raise an obvious question: how can one infer the set of taxa present in an environment from a sample of their DNA? As early as the middle of the 19^{th} century, the father of microbiology, Ferdinand Cohn, proposed a scheme to categorize bacteria to different groups based on their shapes [31]. As more data became available and evolutionary understanding improved, however, scientists realized that cell shape and other phenotypic characteristics were not universally applicable as phylogenetic characters [53]. Adding to the difficulties of microbial taxonomy was the fact that most microbial organisms lack an obligate sexual cycle, so the familiar concept of reproductive isolation considerably less useful as a means to define species. As a result of these difficulties, researchers have sought to use the sequences of these organisms' genes to infer taxonomic definitions (known as Operational Taxonomic Units or OTUs) that are analogous to the species concepts used in classical ecological studies of multicellular organisms. In order to do so, it is first necessary to select a gene or sequence region on which to base these classifications. Among the choices, the gene for 16S subunit of the ribosome has several attractive features. First, the 16S subunit is essential for protein synthesis and a homolog is present in all known organisms. Moreover, the sequence of the 16S subunit is long enough to give a strong phylogenetic signal but short enough

that most of that diversity can be captured in single Sanger sequencing run after PCR amplification from universal primers – e.g., a barcoding approach [53]. The advent of next-generation sequencing techniques [125] only increased the utility of these barcoding approaches, even though the shorter read lengths introduced some minor alterations to protocols: the economics of metagenomics has thus changed drastically since the era of DNA-DNA re-association tests [132].

The state-of-the-art approaches for clustering 16S rDNA samples can be largely divided into two methods: taxonomy-dependent and taxonomy-independent. In taxonomy-dependent methods, the sequences in question are compared against a known and well-annotated database [156]. Taxonomy-independent methods are generally used to analyze large sets of PCR amplicons from an environment that are then compared amongst each other rather to a database. In that case, the sequences in question are assigned to OTUs with pairwise similarity distance information [137] [120], which eliminates the requirement for established taxonomic information and opens the opportunity for the discovery of novel taxa. A well established taxonomy-independent, phenetic method can give accurate estimates of the phylogeny and thus is suitable for inferring relationships between organisms [36] [131]. However, even with a well-selected phylogenetic marker such as the 16S rDNA gene, converting sequence data into taxonomic information is not a trivial problem for two reasons, one practical and one conceptual. Practically, metagenomic samples with thousands of individuals or genes are not particularly computationally tractable given the number of possible phylogenetic relationships between them [49]. The more serious problem is that even a well-resolved phylogeny does not easily distinguish among populations, species and higher taxonomic groups. Given this second issue, it is often simplest to avoid calculating relationships between the groups altogether and instead only attempt to define the taxonomic groups themselves.

Although there are variations in how exactly this OTU classification computation

is performed, the $S \rightarrow D$ (Similarity to Dendogram) approach [131] which employs the power and abundance of the aforementioned 16S subunit sequence data, involves three steps: Sequence alignment, graph construction, and finally sequence/sample clustering. Thus, after the 16S subunit sequences are obtained, those sequences are first compared to determine their pairwise sequence identities. Then these 16S rDNA sequences can be represented as a graph. (As we will discuss, it is also possible to represent the resulting pairwise identities as a distance matrix, but the resulting computations have the same form in either case). In this graph, the nodes are sequences and the edges between nodes are created if two corresponding sequences have identity greater than a threshold. Finally, the graph is clustered in order to determine the OTUs – for instance in the case of single linkage clustering (see below) each OTU corresponds to one partially connected component in the graph.

No matter the clustering method chosen, one still must first select a sequence identity threshold. As there is no clear a priori value of this threshold, historically it was chosen so as to give good agreement with legacy taxonomic classification methods, such as the DNA-DNA association tests [119]. The value that gave such agreement was 3% sequence difference across the 16S gene. It is worth bearing in mind that such agreement does not, on its own, imply that either the prior experiments or the current threshold value represents the population and taxonomic structure of these organisms well. Indeed, other researchers have found that the sequence identity level that defines taxonomic levels varies among different taxonomic groups [120]. The historical nature of this choice of the threshold value is therefore somewhat unsettling for those who think sequence analysis could help revealing a deeper understanding of the species in the microbial world. On the other hand, it is still an open question whether the microbial world actually consists of discrete taxonomic units that can be identified. Cohn has described his thoughts as to the artificial nature of such classification systems [132], doubts which speak to a large controversy as to whether the

microbial world might be a continuum, thus making all clustering methods largely arbitrary in how they select points on that continuum to define taxa [133]. Particularly in non-sexual groups, taxa are human-constructed concepts [131] and are not intrinsic properties of life. As a result, there is the potential for disagreement among researchers as to appropriate thresholds, and there are few quantitative tools for resolving those differences [130].

In this work, we sought to investigate the structure of the 16S rDNA sequence space, and its behavior under differing classification approaches. In contrast to earlier work, our focus is not on particular tools nor even necessarily on defining "the best" clustering method for OTUs. Instead, we were more broadly interested in the degree to which the known set of 16S rDNA sequences naturally fit or do not fit into the framework of numerical taxonomy. We analyzed three factors that might affect OTU inference: 1) the sequence identity threshold, 2) the sampling of the sequence space (e.g., the number of sequences compared) and 3) the clustering method employed. As a baseline, we compared our results across these parameters to reference naming schemes in the NCBI databases. Finally, we explored a number of metrics on the sequence identity graph to assess whether there are an "natural" choices for our identity, sampling and clustering parameters that represent well the underlying nature of the data. Conceptually, such choices would constitute local maxima or minima in property values for some desirable property.

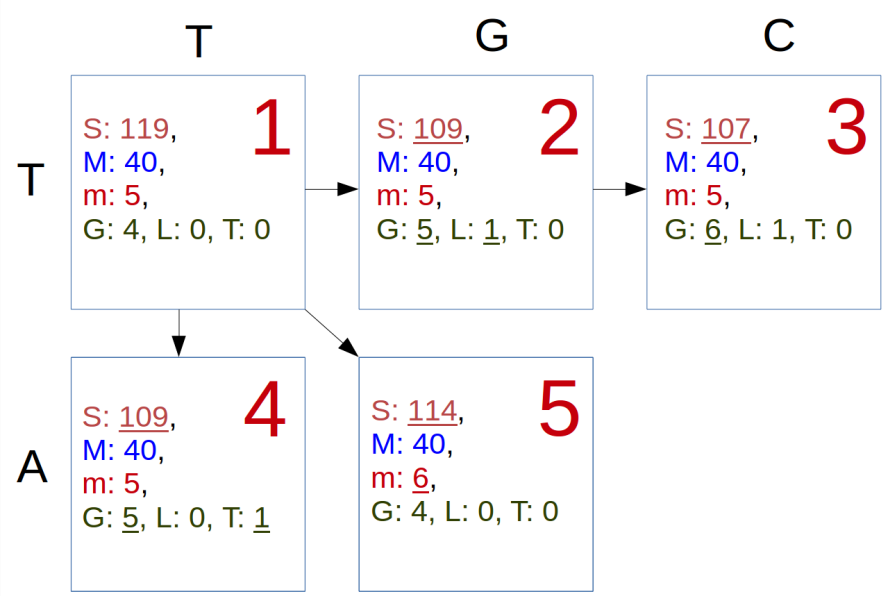


Figure 5.1: The modified Needleman-Wunsch alignment approach to compute sequence identity in GPU. This figure illustrates how adjacent cell scores are computed for path selection consideration given the value of the previous cell. S: Score, M: Number of matches, m: Number of mismatches, G: Number of gaps, L: Horizontal "Left" gap bit, T: Vertical "Top" gap bit. The changed numbers are underlined in each step. Note that the figure only shows examples of calculating the score of one path selection, but does not show the optimal path.

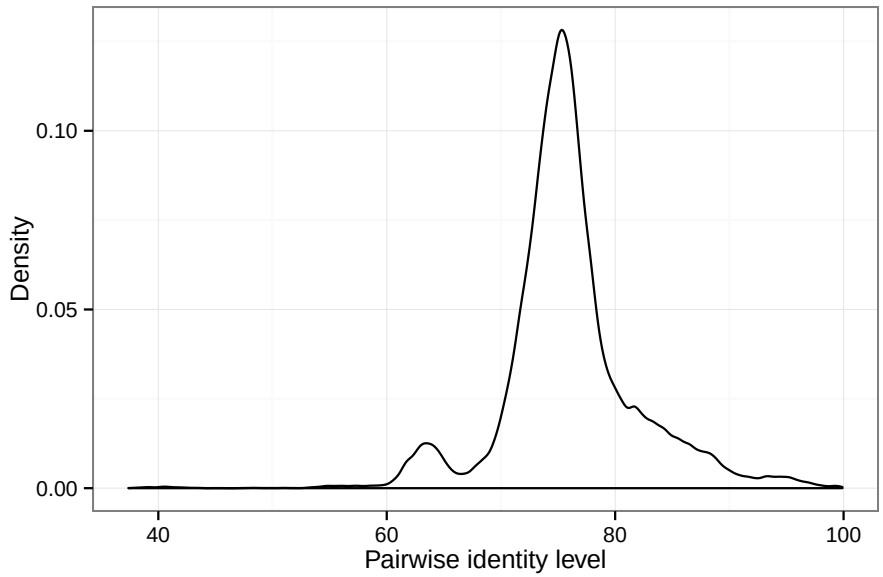


Figure 5.2: Identity levels between pairwise comparisons. We randomly selected 100 sequences among 20,229 sequences, then performed all pairwise comparisons and recorded the results. This experiment was repeated 20 times.

5.2 Results

5.2.1 Sequence identity levels and reference species and genera classification

The first step of our analysis is simply to obtain pairwise differences between the 16S rDNA sequences in the database. However, this problem is not a trivial one: our computational pipeline simplifies it slightly by not explicitly storing identity values for pairs which can be shown to have identity $< 80\%$. All pairs having identity levels below or equal to 80% are assumed to have identity level of 80%. This is to say identity levels below 80% are not considered in further analyses. We believe this trade-off is fair, and it does not affect our results. To estimate the overall structure of this identity distribution, however, we adopted a sub-sampling approach, the results of which are shown in *Figure* 5.2. Most pairs of sequences having identity level between 65% and 75%.

Examining the identity levels in the dataset more closely, we observed a potentially bi-modal distribution. To explore this possibility, we adapted our existing tool for fitting statistical distributions to data by maximum likelihood [108] to fit mixtures of normal distributions to these identity values. We found that a mixture of two normal distributions fit the identity level values better than a single distribution. The bulk of the pairs have sequence identities peaking at around 75%, and a small portions of the pairs having sequence identities peaking at around 65% (See *Figure* 5.2). There seems to be another small peak at 95% identity: however adding a third normal distribution did not significantly improve the fit. Notably, all pairs had higher identity levels than the 25% identity expected for random sequences: exactly the expectation given the 16S rRNA's role in protein synthesis. Nonetheless, the source of these two peaks remains obscure. However, few biologists would argue that such distant evolutionary relationships are a useful metric of microbial ecosystem function. We thus proceeded

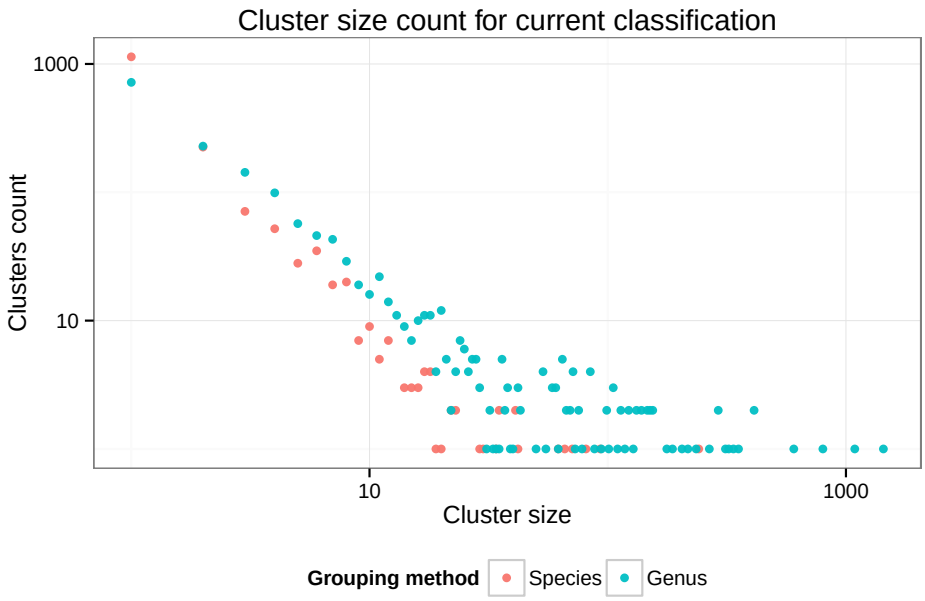


Figure 5.3: Taxonomic unit sizes using current species taxonomic classification as a reference. On the x-axis is the number of sequences assigned to a given taxonomic unit/cluster. On the y-axis is the number of units/clusters of that size found. Results are shown for genera names and species names.

with an analysis of the more similar sequence pairs.

As described in the Methods, our GPU implementation of pairwise sequence alignment only exactly computes sequence identity for pairs of sequences that are greater than 80% identical. This truncation of the computation results in sequence identity network with a maximum of $37,689,969$ edges, which is 80% savings ($\frac{37,689,969}{20,229^2/2}$) from storing all possible pairwise edges. The truncation of pairwise comparisons also enables us to fit the complete network in RAM, accelerating later analyses. As the range of sequence identities of interest to us is from 90% to 100%, those pairs of less than 80% identity cannot possibly affect either single or complete linkage analysis. There is no such guarantee for average linkage analysis, however, we believe the trade-off for accuracy is negligible.

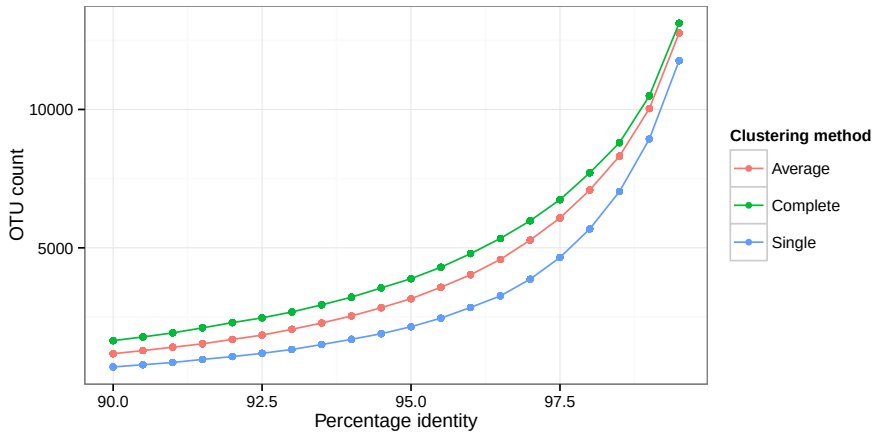


Figure 5.4: Inferred OTU counts as a function of the percentage identity threshold. On the x-axis axis is the percentage identity threshold (e.g., only pairs with that identity or higher were used in OTU construction); on the y-axis is the total number of OTU inferred. Complete clustering gives the highest number of inferred OTUs estimated at every identity threshold (green), followed by average clustering (red) and then single-linkage clustering (blue).

5.2.2 Reference genera and species classifications

Current inferences of the genera and species classifications for the database sequences are illustrated in *Figure* 5.3. The leftmost points are the counts of singleton taxonomic units (e.g., clusters), sequences that are the only known examples of their species or genus. Except for singleton clusters, there are more large clusters when genera classifications are used, as would be expected given that species are defined as subsets of genera.

5.2.3 Tree-like interpretation of the OTU classification

The tree-like interpretation of the pairwise distance values allows us to employ three different clustering algorithms to the sequence data (see Methods). We therefore explored how different clustering algorithms and settings affect OTU inferences and hence the biological implications of those inferences. The analysis was done with our in-house implementation of the UPGMA algorithm with enhancements to allow the

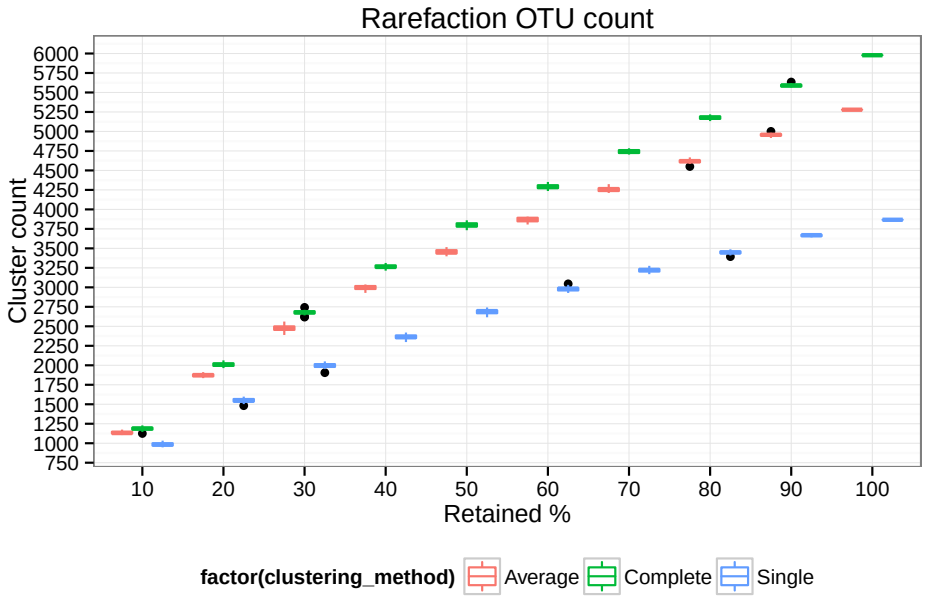


Figure 5.5: Inferred OTU counts as a function of the proportion of the network retained. On the x-axis axis is the percentage of original 16S sequences used for OTU inference; on the y-axis is the total number of OTU inferred. OTU trends are as described for Figure 5.

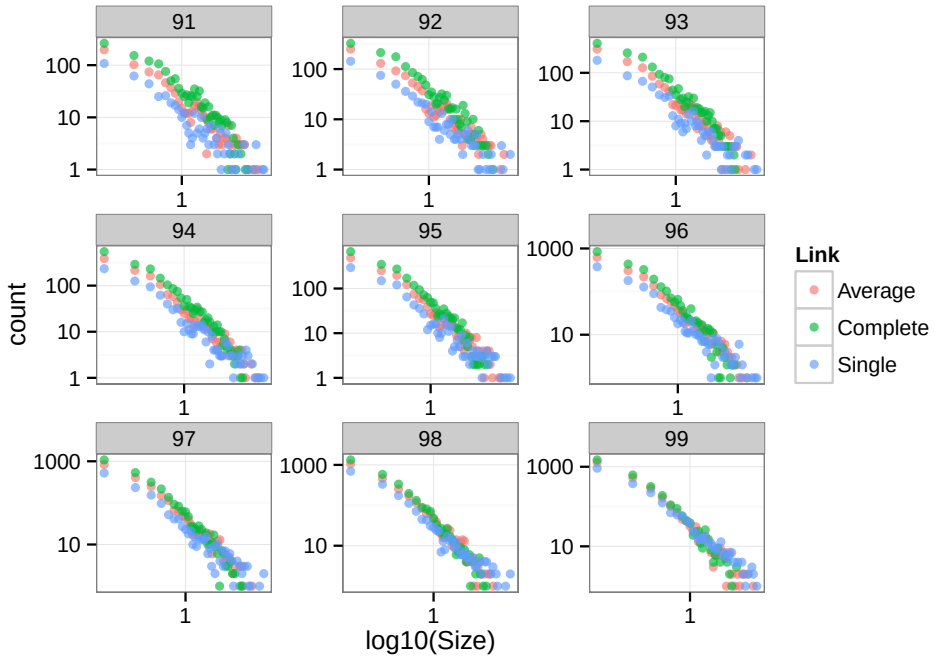


Figure 5.6: OTU Cluster sizes comparison between algorithms and thresholds.

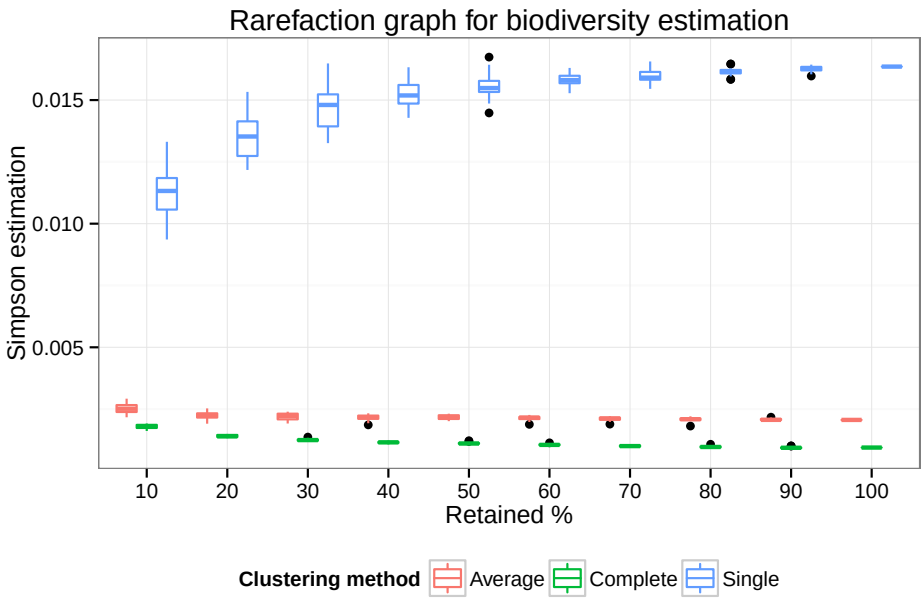


Figure 5.7: Measurement of the Simpson index on different rarefaction levels. On the x-axis axis is the percentage of original 16S sequences used for OTU inference; on the y-axis is Simpson's index for the three clustering methods. Note that the boxplots for average and complete linkage have been slightly shifted for clarity.

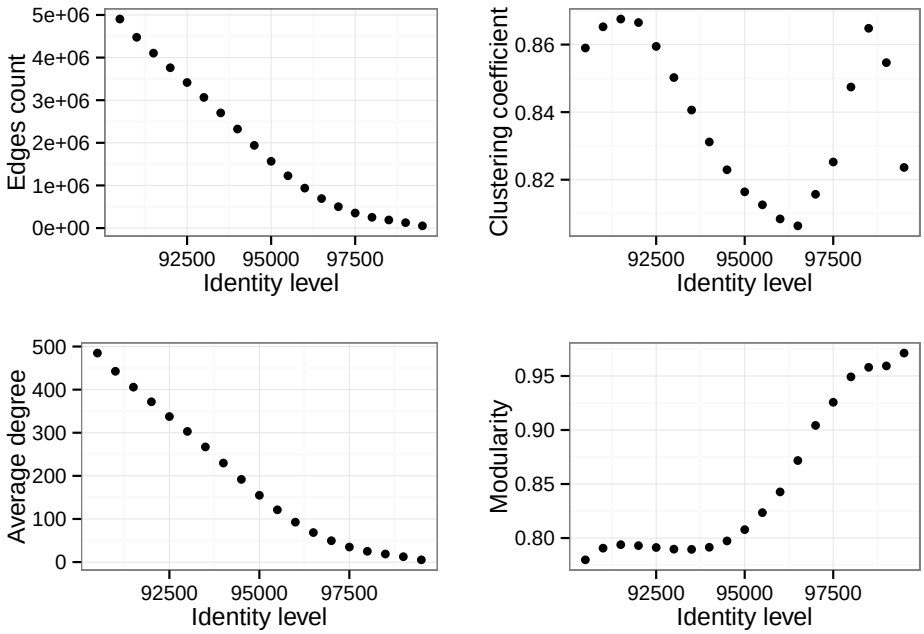


Figure 5.8: Metrics on network under graph representation.

computation of single, average and complete linkage clusters. The program builds a tree-like structure of the resulting OTU network. It starts off by assigning each sequence to its own cluster. The program then iteratively calculates the distances between every cluster in the network according to the linkage algorithm selected. It subsequently merges the eligible clusters (clusters that have distance to each other less than the threshold) until no further eligible clusters are found.

Figure 5.4 shows the total number of OTUs inferred with each approach when the full set of sequences is analyzed. As expected, there is a monotonic increase in the number of OTUs inferred as sequence identity is increased. Likewise, single linkage clustering always infers fewer OTUs than does average linkage clustering, which in turn always infers more than complete linkage clustering. To explore these results in more detail, we plotted the distribution of the OTU sizes in *Figure* 5.6 at a range of percentage identity. At any given identity threshold, complete clustering tends to give more smaller OTUs, while single clustering tends to yield fewer small clusters and more big ones, while average clustering is intermediate between the two. We note that as the threshold decreases from 99% to 90%, the effect of clustering methods is more pronounced: the three curves become more distinct. The difference in the abundances of singleton clusters across different thresholds is stark: While there are only in the order of hundreds singleton clusters in the case of 91% identity level, there are thousands at a threshold of 99% . Thus, a more stringent sequence identity threshold may tend to suggest the existence of rare taxa over more relaxed thresholds.

The trends seen for the complete sequence dataset are of course expected. What is less well understood is the relative performance of the three algorithms when the dataset is incomplete. With a fixed percentage identity setting of 97%, we investigated how repeated subsampling affects the final OTU count. *Figure* 5.5 shows how the OTUs estimates vary for partial networks. Again, not unexpectedly, all three methods infer more clusters when more sequences are considered, but the rate of increase in

OTU number decays as more sequences are added. It was initially surprising to us that none of the three methods seemed to plateau as new sequences are added. However, we realized that our dataset was not repeated samples from the same environment and hence is not expected to display such behavior. In environmental samples, rare microorganisms indeed contribute to the long-tailed composition of microbiomes thus increase the amount of unseen OTUs, for example, it is known while the human gut consists of 500 to 1000 "species" [52], 30 to 40 dominant "species" makes up 99% of the samples [117] [113] ("species" as in the context of the sources). However, every such species is represented only by its unique 16S rDNA sequences in the database, giving rise to a different abundance structure that is less well understood.

Overall, these data show two other relevant and novel features. First, partial sampling affects single clustering much less than is the case for complete linkage clustering. Second, the variance between random resamples are surprisingly low across resampling levels. This lack of variance indicates that randomness in obtaining sequence samples does not significantly affect at least the final OTU count (though it may affect the particular OTU found). This observation would tend to suggest at least some structure to the sequence data, such that OTU are stable definitions even in the face of partial sequence sampling.

Sampling potentially affects OTU inference in two ways. First, it can obviously split clusters if a key sequence is missing. More subtly, the set of available edges may alter the order in which clusters are merged. To explore this effect, we computed the mean number of named genera and species (see above) found at each subsampling level as well as the inferred number of OTU under the three clustering schemes at different identity thresholds. We then measured different properties of the inferred OTUs themselves. First, we looked at how clustering method and identity level affects estimation of OTU. Then, we looked at how sampling/rarefaction changes the OTU inferences. Finally, we explored the biodiversity implications of such clustering

approaches by looking at the diversity richness measurements of the dataset given the algorithms.

We also compared the network rarefaction with the database-derived genera and species classification to see how subsampling changes the biological diversity observed. Besides the number of OTUs, the Shannon-Wiener index [124] and Simpson index [126] are two important measures of biodiversity which we applied to these data. The Shannon index measures whether the OTUs sizes are evenly distributed, and is given by $H = -sum(P_i \times ln[P_i])$. The Simpson index measures the probability that two randomly selected individuals belonging to the same OTU and is given by: $D = sum(P_i^2)$ where P_i is the probability of observing a sample belonging to a particular OTU. In our investigation, we did not discover any significant difference in the inferred Shannon indices between the three algorithms (See Suppl. Figure 2). However, at any given rarefaction level, the Simpson index is significantly higher in the single linkage case compared to the other two methods (See *Figure* 5.7).

5.2.4 Network-based analysis of OTU classification approaches

As discussed before, a network representation of these data corresponds exactly to the single linkage clustering method above. Under in this graph-theoretic framework, we were able to explore additional properties of the network that were not obvious when merely using distance matrices. *Figure* 5.8 shows four network metrics on the OTU graph, under 100% network retention with varying identity level thresholds.

The total edge number and average degree give, respectively, the number of connected nodes (e.g., pairs of 16S genes with sequence identity greater than the threshold), and the average number of edges one node possesses. Both statistics monotonically decrease as the identity threshold increases, reaching zero when the percent identity reaches 100%. The clustering coefficient and modularity of the networks show more complex changes as the percentage threshold changes.

In particular, the clustering coefficient, which is often thought of as a measure of "cliquishness" in a network, actually reaches a minimum near the commonly used percentage identity threshold of 97%. The clustering coefficient is defined as the number of fully connected triplets of nodes divided by the number of total connected triplets. At first blush, this result is discouraging, as well-defined clusters in the network should yield high clustering coefficients. However, we suspect that this result is partly driven by the fact that isolated network nodes have undefined clustering coefficients and as the percentage identity nears 100%, more and more sequences will be isolated (have no edges), altering the structure of the network.

5.3 Discussion

It would be ideal if microbial genomes included convenient flags that uniquely distinguished differing groups of bacteria or archea, allowing rapid and inexpensive sequence-based classification. Our analyses of the set of known prokaryotic 16S rDNA genes suggests that the problem is more complex than that. In particular, while differing clustering algorithms yield different groups of OTUs at different identity thresholds, there is no obvious reason for preferring one to another. While this conclusion is slightly disheartening, further study of figures 5.5 and 5.7 is slightly more encouraging, since they suggest that within reasonable bounds, the methods and thresholds do not actually yield terribly different estimates.

Our analysis of course has a number of limitations. Most obviously, we have worked only with the set of known 16S rDNA genes. This set is biased towards known environments and taxa and likely does not fully reflect diversity among microbes. Thus, while we present a back-of-the-envelop computation of the saturation of the current database, the value given is almost certainly an underestimate. Moreover, because the database we employed has these biases, its behavior across sequence

identity levels and sub-sampling may not be reflective of how an unbiased sample would behave. Since our primary goal, however, is to understand the interaction of clustering approaches and dataset size, many of our conclusions are likely not highly sensitive to these issues. More seriously, the database only represents unique sequences, giving no indication of population size and structure. Since population size alters rates of evolution and polymorphism levels in the population [158] [27], one potential reason that percentage identity methods are inconsistent (as seen above) is that different populations with differing sizes undergo different evolutionary dynamics in their 16S rDNA genes.

Perhaps the most general conclusion of our analysis is that, within the framework of pairwise alignment and clustering, while algorithm choices will alter one's results, there are not obviously superior and inferior choices among these algorithms. As can been seen in *Figure* 5.4, the three different clustering algorithms do not enormously differ in their set of inferred OTUs. This result was surprising to us, as we anticipated that the single linkage clustering would give much lower OTU counts due to its tendency to connect two previously separated OTUs together with the addition of just one new connection. This trend, however, is not pronounced in the actual data: the curve from single-linkage clustering is similar to other clustering algorithms. Similarly, it does not appear that any particular choice of identity threshold maps "naturally" on to the known set of 16S rDNA sequences, although the range 96-97% identity does present some interesting features with respect to the network clustering coefficient. This somewhat encouraging conclusion can be combined with the fact that many of the most illuminating metagenomic studies have focused less on the presence or absence of individual taxa and more on questions such as the structure of the ecosystems involved [66] [67] [136]

If the various clustering approaches yield similar results, is there any reason for preferring one over another? Previous studies suggest that average linkage clustering

might be the most stable approach [131], with the extreme of single and complete linkage clustering being less preferable. While we also find that average linkage gives intermediate results, its performance is not radically different from single linkage clustering. However, both average and complete linkage clusters are significantly more computationally expensive to compute than are single linkage clusters. On its own, this fact is relatively unimportant, as the major computational cost of all three techniques is the all-against-all pairwise comparisons performed here with the GPU. However, single linkage clustering has the added advantage of also being representable in a graph framework. This framework, in addition to slightly increased computational efficiency, allows one to more thoroughly explore the nature of the clusters through the use of the types of graph statistics considered here. While our initial analysis did not find that these statistics greatly altered our view of the clustering problem, we hope that other analyses in the future will provide greater insights. In sum, the arguably most common current practice of computing OTUs via single-linkage clustering at a 97% identity cutoff, while perhaps not theoretically highly justified, appears to provide performance similar enough to other avalaible methods as to make it reasonable to continue to use it as a standard.

5.4 Methods

5.4.1 Data source and reference classification analysis

The database of 16S rDNA subunit genes used was created by merging the sequences in the Ribosomal Database collection [23] and the set of 16S prokaryotic genomes of NCBI GenBank [10]. From these sequences, we eliminated all identical sequences as well as any sequences less than 1450 bases long or with unidentified bases (no "N" character states were allowed). Our resulting sequence dataset consists of 20,229 se-

quences. The mean length of these sequences is 1502 bases (Supplemental Figure 1). For reference, we extracted the current genera and species classifications for each sequence from the FASTA header lines. For 112 sequences with unknown or unspecified genera, we assigned a new genus name to each sample. Doing so significantly simplifies the analysis pipeline as we have the same number of samples in all comparisons, with the trade-off of inflating the number of genera by at worst 112 singleton genera (among 719 singleton genera).

5.4.2 All-against-all sequence alignments on GPU and sequence identity computation

As a first step to analyzing the taxonomic structure of these sequences, we chose to compute all-against-all pairwise sequence alignments to obtain the percent sequence identity of all pairs in the dataset. We chose not to perform the alternative multiple alignment computation as previous studies have pointed out that pairwise alignment gives a better estimation of the true OTUs [57]. This pairwise approach requires that $204,596,106$ comparisons (and hence alignments) be carried out.

To perform this somewhat daunting computation, we employed our in-house GPU-based program on the CUDA architecture [97], based on the $LazyRScan - mNW$ Needleman-Wunsch global alignment method in linear-memory in [146]. From these comparisons, we recorded all sequences that are more than 80% identical at 0.1% precision.

To understand how the program carries out the computation, we must recall how a standard Needleman-Wunsch alignment is performed. To align two sequences of length m and n respectively, the program first conceptually creates a matrix of size $(m+1)*(n+1)$. Then it sequentially fills in each cell (i,j) in the matrix with the optimal cost to get to that cell from the beginnings of the two sequences. Many implementations forgo the full $(m+1)*(n+1)$ matrix allocation and store only a

portion of that matrix. However, all approaches must visit the $(m+1)*(n+1)$ cells in that conceptual matrix in their computation. By forgoing the storage of the full matrix, these implementations also must use various recursive strategies to recover the sequence alignment itself from the partial score matrix that is stored. At the atomic level, the process of calculating the alignment score involves selecting the highest scoring path among three possible choices. The three choices to move from one cell from the previous ones are: Creating either a gap in sequence 1, a gap in sequence 2, or aligning the two bases of the sequences.

$$F_{i,j} = \max(\begin{cases} F_{i-1,j-1} + sub(A_i, B_j) \\ F_{i,j-1} + cost\_gap \\ F_{i-1,j} + cost\_gap \end{cases})$$

For:

$$sub(A_i, B_j) = \begin{cases} cost\_match \text{ if } A_i = B_j \\ cost\_mismatch \text{ if } A_i \neq B_j \end{cases}$$

Our implementation, referred to as $LazyRScan - mNW$, significantly speeds up the computation of the optimal score of the alignment by allowing a very large number of alignment scores to be computed in parallel on a graphics processing unit (GPU). To do so, we reframed the computation to fit into the relatively small fast cache on the GPU, placing each vertical slice consisting only k columns of the whole calculation into small constant-sized memory slices. The reason for this division is that the GPU cache is not nearly large enough to hold the ˜1500×1500 element matrix in the naive memory implementation. This fixed slice of fast cache is repeatedly recycled $1500/k$ times as the computation progresses until the computation reaches the last cell. While this approach allows the computation of the pairwise alignment score very efficiently in parallel, it makes the recovery of the sequence alignment itself (and

hence the alignment percent identity) problematic. To address this limitation, we developed several important enhancements to the $LazyRScan - mNW$ method for this work. These enhancements allows the GPU to calculate the exact identity level in one pass on the GPU, without having to do a slower traceback step or recursive calculations on the CPU to recover the actual alignment. Our method is based on the realization that the identity level in the optimal alignment can be calculated without knowing the actual alignment. Recall that necessarily the percent identity of the alignment is given by:

$$identity\_level = \frac{matches}{matches + mismatches + gaps}$$

Thus, by recording the matches, mismatches, and gaps at each step, we are guaranteed to know the identity level of two sequences without the knowledge of how exactly the two sequences align. We retain those four attributes at every cell of the alignment matrix computation in the form of a 3-tuples: $(score, matches, mismatches)$. By keeping track of these numbers in each path until the final cell of the matrix, quickly calculating the identity level of the alignment is possible.

Since the introduction of the gap on each sequence is highly undesirable while the extension of the gap in the same sequence is not as bad, we also introduce two bits indicating whether the cell can potentially extend a gap vertically and horizontally in the optimal path ("gap bit"). In our CUDA implementation, the narrowest type that can carry out numerical operations is a 16-bit short integer. The gap bits piggyback on the *gaps* field, so that we do not waste the memory needed for dedicated fields for the bits. This data-packing limits the maximum number of gaps an alignment can have to $16,383$ (14-bit) instead of $65,535$ (16-bit). Both of the horizontal and vertical gap bits are initially set to 0, in which case an introduction of a gap adds the gap open penalty to the score and sets the respective gap bit to 1. Any match or mismatch alignment closes the gap, and sets the gap bits back to 0. For this project,

we used the following scoring matrix: Match = +4, Mismatch = -5, Gap open = -10, Gap extension = -2. The optimal alignment score calculation process is illustrated in *Figure* 5.1.

5.4.3 Building OTU clusters from pairwise alignments

After all pairwise alignment percent identities are computed, we implement the inference of OTUs as a clustering computation on a conceptual $N \times N$ similarity matrix of percentage identities. Initially each 16S rDNA gene is placed into its own cluster. Then clusters are progressively joined into larger clusters when certain conditions are met, depending on the algorithm. This approach is algorithmically very similar to how a phenetic tree is constructed. Here, we have evaluated three methods for clustering percent identity data among sequences:

- **Complete linkage**: All sequences in one cluster are guaranteed to be more than k% identical to every other sequence in the cluster (farthest neighbor).
- **Average linkage**: All sequences in one cluster are guaranteed to be *on average* more than k% identical to every other sequence in the cluster (Unweighted Pair Group Method with Arithmetic Mean) [44].
- **Single linkage**: All sequences in one cluster are guaranteed to have at least one pair of sequences in each cluster that is more than k% identical (nearest neighbor).

As the percent identity is reduced, each clustering approach will join more sequences into the same cluster, meaning that the total number of inferred clusters (or OTUs) will decrease. In addition to thus exploring the role of percent identity in OTU inference, we also assessed the effect of partial sampling of sequences in an environment. We did so by randomly subsampling the set of sequences in our database at 10% increments, with 1000 re-samplings at each increment.

Interestingly, in parallel with matrix/tree-like approaches, the OTU clustering problem can also be represented as a graph problem. In this graph, each 16 rDNA gene is a node, and edges connect nodes when their sequence identity exceeds a preset threshold. OTUs are then defined as connected components of the graph (e.g., the connected components in the graph are identical to the clusters in single linkage clustering). Representing the problem as a graph, however, allows us to analyze the the clusters with graph-based metrics not available from tree or distance-based approaches. We considered the following graph metrics: average degree, clustering coefficient, modularity and the density of the graph.

An overview of our approach on analyzing the OTU estimation is shown on Appendix *Figure A*.1.

5.5 Conclusion

In the study, we have investigated on different OTU classification methods of microorganisms based on the 16S sequence identity their respective sequences. We have proposed a high-performance method to quickly calculate all possible pairwise distances for large datasets using a GPU. We then investigated different clustering methods. We found out that no single method or setting is clearly preferable to the others. With the knowledge that having secondary structures do not necessarily improve the quality of the OTU assignment [156], this result makes OTU estimation a difficult call for microbiologists. It is possible that the the nature of microbial population structure is too subtle to represent with such crude computations [27]. On the other hand, even crude models may give sufficient resolution for ecosystem-type models, in which case we proposed that single linkage may be no less efficient than slower and more sophisticated methods.

Chapter 6

Summary

In this dissertation, I have presented two important algorithms to accelerate homology sequence analysis using the NVIDIA GPU platform: Sequence alignment and k-mer composition comparison. I have also presented the application of such algorithms in answering two questions in biology: Upstream comparison leading to coregulation pattern discovery, as well as OTU definition.

The implications of the work are as follows:

- First, the solutions for both the sequence pairwise comparison and motif mining problem are truly high-performance solutions. The computation carries out an exact and exhaustive search with results comparable to the equivalent CPU code, instead of relying on heuristics to eliminate computational work. This elimination of heuristics avoids the problem of potentially having false negative results (failure to identify a true sequence match). In addition, it also implies further optimizations to the heuristics could further expand the performance of the pipeline.

- Second, the solution can be integrated into newer hardware and other pipelines, making it easy to stream through additional data to improve the answer. This

provides a new venue for advances in sequencing technology, in such the answer is improved by having more data, instead of better algorithms.

- Third, contributing to basic computer science research, the solution is applicable to other string manipulation problems, in particular, approximate string matching. Carrying on this operation quickly is crucial in many applications, including text correction, search suggestion [164], 3D image reconstruction via pixel-matching [51], and even reducing bandwidth for the delivery of patches for mobile applications.

- Fourth, as silicon shrinking progress is slowing down and problems start lurking in making bigger cores, having smaller cores in GPUs is a viable path to discovery in biology. As a bonus, the accessibility and availability of such GPUs in consumer hardware means that ordinary citizens can participate in science by verifying claims, contributing computational power, and making new discoveries.

Appendix A

OTU Analysis: OTU estimation pipeline

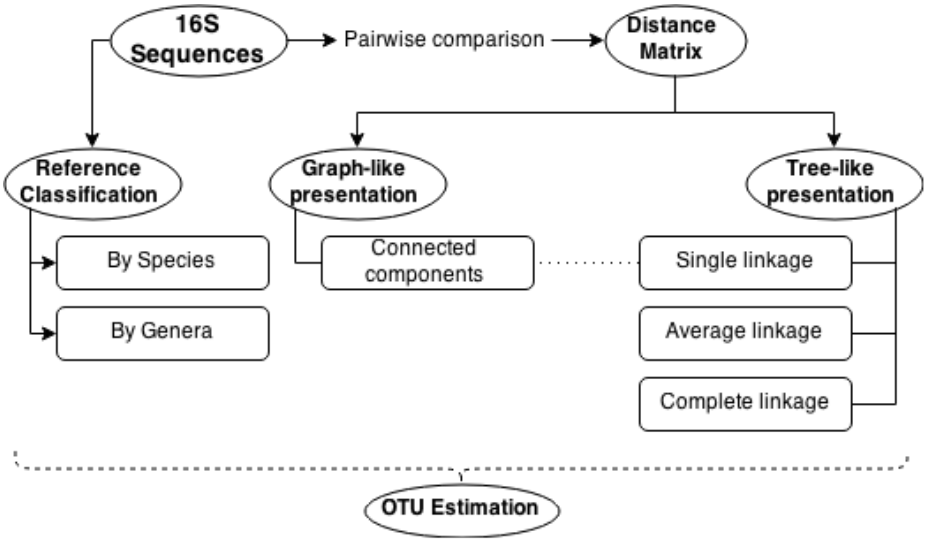


Figure A.1: OTU estimation pipeline from 16S sequence samples. All three major methods lead to a particular estimation of the OTU structure of the samples. The single linkage clustering in the tree-like aprach gives the same result as counting partially connected components in graph-like approach.

Bibliography

[1] Bronwen L. Aken et al. "Ensembl 2017". In: *Nucleic Acids Research* 45.D1 (2017), pp. D635–D642. DOI: 10.1093/nar/gkw1104. eprint: /oup/backfile/content_public/journal/nar/45/d1/10.1093_nar_gkw1104/3/gkw1104.pdf. URL: http://dx.doi.org/10.1093/nar/gkw1104.

[2] Dan A Alcantara et al. "Building an efficient hash table on the GPU". In: *GPU Computing Gems* 2 (2011), pp. 39–53.

[3] Dan A. Alcantara et al. "Real-time Parallel Hashing on the GPU". In: *ACM Trans. Graph.* 28.5 (Dec. 2009), 154:1–154:9. ISSN: 0730-0301. DOI: 10.1145/1618452.1618500. URL: http://doi.acm.org/10.1145/1618452.1618500.

[4] S. F. Altschul et al. "Basic Local Alignment Search Tool". In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410.

[5] S. F. Altschul et al. "Gapped Blast and Psi-Blast : A new-generation of protein database search programs". In: *Nucleic Acids Research* 25.17 (1997), pp. 3389–3402.

[6] 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 International Symposium on*, pp. 1–12. ISBN: 1530-2075. DOI: 10.1109/IPDPS.2008.4536261.

[7] K. Balhaf et al. "Using GPUs to speed-up Levenshtein edit distance computation". In: *2016 7th International Conference on Information and Communication Systems (ICICS)*. Apr. 2016, pp. 80–84. DOI: 10.1109/IACS.2016.7476090.

[8] Michela Becchi and Patrick Crowley. "Efficient regular expression evaluation: theory to practice". In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, pp. 50–59. ISBN: 1605583464.

[9] Oded Béjà et al. "Unsuspected diversity among marine aerobic anoxygenic phototrophs". In: *Nature* 415.6872 (2002), p. 630.

[10] Dennis A Benson et al. "GenBank". In: *Nucleic acids research* (2012), gks1195.

[11] Dennis Benson, David J Lipman, and James Ostell. "GenBank". In: *Nucleic Acids Research* 21.13 (1993), pp. 2963–2965.

[12] Andreas Biegert et al. "The MPI Bioinformatics Toolkit for protein sequence analysis". In: *Nucleic Acids Research* 34 (2006), p. 5. DOI: 10.1093/nar/gkl217.

[13] Ewan Birney et al. "Identification and analysis of functional elements in 1% of the human genome by the ENCODE pilot project". In: *Nature* 447.7146 (2007), pp. 799–816.

[14] Mathieu Blanchette, Benno Schwikowski, and Martin Tompa. "Algorithms for phylogenetic footprinting". In: *Journal of computational biology* 9.2 (2002), pp. 211–223.

[15] Paola Bonfante and Iulia-Andra Anca. "Plants, mycorrhizal fungi, and bacteria: a network of interactions". In: *Annual review of microbiology* 63 (2009), pp. 363–383.

[16] Kevin P Byrne and Kenneth H Wolfe. "The Yeast Gene Order Browser: combining curated homology and syntenic context reveals gene fate in polyploid species". In: *Genome research* 15.10 (2005), pp. 1456–1461.

[17] Niccolo' Cascarano et al. "iNFAnt: NFA pattern matching on GPGPU devices". In: *ACM SIGCOMM Computer Communication Review* 40.5 (2010), pp. 20–26. ISSN: 0146-4833.

[18] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee. 2009, pp. 44–54.

[19] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[20] Andrew G Clark et al. "Evolution of genes and genomes on the Drosophila phylogeny". In: *Nature* 450.7167 (2007), pp. 203–218.

[21] Paul Cliften et al. "Finding functional features in Saccharomyces genomes by phylogenetic footprinting". In: *science* 301.5629 (2003), pp. 71–76.

[22] Joel E Cohen. "Mathematics is biology's next microscope, only better; biology is mathematics' next physics, only better". In: *Plos biol* 2.12 (2004), e439.

[23] James R Cole et al. "The Ribosomal Database Project: improved alignments and new tools for rRNA analysis". In: *Nucleic acids research* 37.suppl 1 (2009), pp. D141–D145. ISSN: 1362-4962 (Electronic) 0305-1048 (Linking). DOI: gkn879[pii] 10.1093/nar/gkn879. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=19004872.

[24] Gavin C Conant. "Comparative genomics as a time machine: how relative gene dosage and metabolic requirements shaped the time-dependent resolution of yeast polyploidy". In: *Molecular biology and evolution* 31.12 (2014), pp. 3184–3193.

[25] Human Microbiome Project Consortium et al. "A framework for human microbiome research". In: *Nature* 486.7402 (2012), pp. 215–221.

[26] Sara J Cooper et al. "Comprehensive analysis of transcriptional promoter structure and function in 1% of the human genome". In: *Genome research* 16.1 (2006), pp. 1–10.

[27] Otto X Cordero and Martin F Polz. "Explaining microbial genomic diversity in light of evolutionary ecology". In: *Nature Reviews Microbiology* 12.4 (2014), p. 263.

[28] Moheb Costandi. "Citizen microbiome". In: *Nature biotechnology* 31.2 (2013), pp. 90–90.

[29] Modan K. Das and Ho-Kwok Dai. "A survey of DNA motif finding algorithms". In: *BMC Bioinformatics* 8.7 (2007), pp. 1–13. ISSN: 1471-2105. DOI: 10.1186/1471-2105-8-s7-s21. URL: http://dx.doi.org/10.1186/1471-2105-8-S7-S21.

[30] P. Dlugosch et al. "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing". In: *Parallel and Distributed Systems, IEEE Transactions on* 25.12 (2014), pp. 3088–3098. ISSN: 1045-9219. DOI: 10.1109/TPDS.2014.8. URL: http://ieeexplore.ieee.org/ielx7/71/6951447/06719386.pdf?tp=&arnumber=6719386&isnumber=6951447.

[31] Gerhart Drews. "Ferdinand Cohn, a founder of modern microbiology". In: *ASM News* 65.8 (1999), p. 547.

[32] Casey W Dunn, Xi Luo, and Zhijin Wu. "Phylogenetic analysis of gene expression". In: *Integrative and comparative biology* 53.5 (2013), pp. 847–856.

[33] Sean R Eddy et al. "Where did the BLOSUM62 alignment score matrix come from?" In: *Nature biotechnology* 22.8 (2004), pp. 1035–1036.

[34] Melinda J Ellison et al. "Diet Alters Both the Structure and Taxonomy of the Ovine Gut Microbial Ecosystem". In: *DNA Research* (2013), dst044.

[35] Peggy J Farnham. "Insights from genomic profiling of transcription factors". In: *Nature Reviews Genetics* 10.9 (2009), pp. 605–616. ISSN: 1471-0064 (Electronic) 1471-0056 (Linking). DOI: nrg2636[pii]10.1038/nrg2636. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=19668247.

[36] James S Farris. "Estimating phylogenetic trees from distance matrices". In: *American Naturalist* (1972), pp. 645–668.

[37] Joseph Felsenstein. "Phylogenies and the comparative method". In: *The American Naturalist* 125.1 (1985), pp. 1–15.

[38] Paul Flicek et al. "Ensembl 2013". In: *Nucleic acids research* 41.Database issue (2012), gks1236. ISSN: 1362-4962 (Electronic) 0305-1048 (Linking). DOI: gks1236[pii]10.1093/nar/gks1236. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=23203987.

[39] Rute R. da Fonseca et al. "Next-generation biology: Sequencing and data analysis approaches for non-model organisms". In: *Marine Genomics* 30 (2016), pp. 3–13. ISSN: 1874-7787. DOI: https://doi.org/10.1016/j.margen.2016.04.012. URL: http://www.sciencedirect.com/science/article/pii/S1874778716300368.

[40] Stoyan Georgiev et al. "Evidence-ranked motif identification". In: *Genome biology* 11.2 (2010), R19.

[41] Sara Goodwin, John D McPherson, and W Richard McCombie. "Coming of age: ten years of next-generation sequencing technologies". In: *Nature Reviews Genetics* 17.6 (2016), p. 333.

[42] Jonathan L Gordon, Kevin P Byrne, and Kenneth H Wolfe. "Additions, losses, and rearrangements on the evolutionary route from a reconstructed ancestor to the modern Saccharomyces cerevisiae genome". In: *PLoS genetics* 5.5 (2009), e1000485.

[43] T. R. Gregory. "Coincidence, coevolution, or causation? DNA content, cell size, and the C-value enigma". In: *Biol Rev Camb Philos Soc* 76 (2001), pp. 65–101.

[44] Ilan Gronau and Shlomo Moran. "Optimal Implementations of UPGMA and Other Common Clustering Algorithms". In: *Inf. Process. Lett.* 104.6 (Dec. 2007), pp. 205–210. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2007.07.002. URL: http://dx.doi.org/10.1016/j.ipl.2007.07.002.

[45] Michael J Hallock et al. "Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations". In: *Parallel computing* 40.5 (2014), pp. 86–99.

[46] C. T. Harbison et al. "Transcriptional regulatory code of a eukaryotic genome". In: *Nature* 431.7004 (2004), pp. 99–104. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=15343339.

[47] Christopher T Harbison et al. "Transcriptional regulatory code of a eukaryotic genome". In: *Nature* 431.7004 (2004), p. 99.

[48] Ross C Hardison and James Taylor. "Genomic approaches towards finding cis-regulatory modules in animals". In: *Nature Reviews Genetics* 13.7 (2012), pp. 469–483.

[49] David M Hillis et al. *Molecular systematics*. Vol. 23. Sinauer Associates Sunderland, MA, 1996.

[50] Daniel S. Hirschberg. "A linear space algorithm for computing maximal common subsequences". In: *Communications of the ACM* 18.6 (June 1975), pp. 341–343. ISSN: 0001-0782. DOI: 10.1145/360825.360861. URL: http://doi.acm.org/10.1145/360825.360861.

[51] Heiko Hirschmuller. "Semi-global matching-motivation, developments and applications". In: *Photogrammetric Week 11* (2011), pp. 173–184.

[52] Lora V Hooper and Jeffrey I Gordon. "Commensal host-bacterial relationships in the gut". In: *Science* 292.5519 (2001), pp. 1115–1118.

[53] John L Howland. *The surprising Archaea: discovering another domain of life*. Oxford University Press, 2000.

[54] Yifan Hu. "Efficient, high-quality force-directed graph drawing". In: *Mathematica Journal* 10.1 (2005), pp. 37–71.

[55] Andrew Huang. "Moore's Law is Dying (and that could be good)". In: *IEEE Spectrum* 52.4 (2015), pp. 43–47.

[56] Timothy R Hughes et al. "Functional discovery via a compendium of expression profiles". In: *Cell* 102.1 (2000), pp. 109–126.

[57] Susan M. Huse et al. "Ironing out the wrinkles in the rare biosphere through improved OTU clustering". In: *Environmental Microbiology* 12.7 (2010), pp. 1889–1898. ISSN: 1462-2920. DOI: 10.1111/j.1462-2920.2010.02193.x. URL: http://dx.doi.org/10.1111/j.1462-2920.2010.02193.x.

[58] Oleksandr Kalentev et al. "Connected component labeling on a 2D grid using CUDA". In: *Journal of Parallel and Distributed Computing* 71.4 (2011), pp. 615–620. ISSN: 0743-7315. URL: http://www.sciencedirect.com/science/article/pii/S0743731510002108.

[59] Manolis Kellis et al. "Sequencing and comparison of yeast species to identify genes and regulatory elements". In: *Nature* 423.6937 (2003), p. 241.

[60] M. Kellis et al. "Sequencing and comparison of yeast species to identify genes and regulatory elements". In: *Nature* 423 (2003), pp. 241–254.

[61] M. Kim, M. Morrison, and Z. Yu. "Status of the phylogenetic diversity census of ruminal microbiomes". In: *FEMS Microbiol Ecol* 76.1 (2011), pp. 49–63. ISSN: 1574-6941 (Electronic) 0168-6496 (Linking). DOI: 10.1111/j.1574-6941.2010.01029.x. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=21223325.

[62] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203.

[63] R Daniel Kortschak and David L Adelson. "biogo: a simple high-performance bioinformatics toolkit for the Go language". In: *bioRxiv* (2014). DOI: 10.1101/005033. eprint: https://www.biorxiv.org/content/early/2014/05/12/005033.full.pdf. URL: https://www.biorxiv.org/content/early/2014/05/12/005033.

[64] B. Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome Biology* 10.3 (2009), R25. ISSN: 1465-6914 (Electronic) 1465-6906 (Linking). DOI: gb-2009-10-3-r25[pii] 10.1186/gb-2009-10-3-r25. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=19261174.

[65] Sylvain Lefebvre and Hugues Hoppe. "Perfect Spatial Hashing". In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 579–588. ISSN: 0730-0301. DOI: 10.1145/1141911.1141926. URL: http://doi.acm.org/10.1145/1141911.1141926.

[66] Roie Levy and Elhanan Borenstein. "Metabolic modeling of species interaction in the human microbiome elucidates community-level assembly rules". In: *Proceedings of the National Academy of Sciences* 110.31 (2013), pp. 12804–12809.

[67] Ruth E Ley et al. "Worlds within worlds: evolution of the vertebrate gut microbiota". In: *Nature Reviews Microbiology* 6.10 (2008), pp. 776–788.

[68] D. Li et al. "A distributed CPU-GPU framework for pairwise alignments on large-scale sequence datasets". In: *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*. June 2013, pp. 329–338. DOI: 10.1109/ASAP.2013.6567598.

[69] H. Li, J. Ruan, and R. Durbin. "Mapping short DNA sequencing reads and calling variants using mapping quality scores". In: *Genome Research* 18.11 (2008), pp. 1851–8. ISSN: 1088-9051 (Print) 1088-9051 (Linking). DOI: gr.078212.108[pii]10.1101/gr.078212.108. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=18714091.

[70] Junjie Li, Sanjay Ranka, and Sartaj Sahni. "Pairwise sequence alignment for very long sequences on GPUs". In: *International Journal of Bioinformatics Research and Applications 2* 10.4-5 (2014), pp. 345–368.

[71] Kuo-Bin Li. "ClustalW-MPI: ClustalW analysis using distributed and parallel computing". In: *Bioinformatics* 19.12 (2003), p. 2. DOI: 10.1093/bioinformatics/btg192.

[72] Chaim Linhart, Yonit Halperin, and Ron Shamir. "Transcription factor and microRNA motif discovery: the Amadeus platform and a compendium of metazoan target sets". In: *Genome research* 18.7 (2008), pp. 1180–1189.

[73] Moritz Lipp et al. "Meltdown". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[74] Chi-Man Liu et al. "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads". In: *Bioinformatics* 28.6 (2012), pp. 878–879.

[75] Hsi Liu et al. "New tests for syphilis: rational design of a PCR method for detection of Treponema pallidum in clinical specimens using unique regions of the DNA polymerase I gene". In: *Journal of clinical microbiology* 39.5 (2001), pp. 1941–1946.

[76] W. Liu et al. "Streaming algorithms for biological sequence alignment on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 19 (2007), pp. 1270–1281.

[77] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. "CUDASW++: Optimizing Smith-Waterman Sequence Database Searches for CUDA-enabled Graphics Processing Units". In: *BMC Research Notes* 2.73 (2009).

[78] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions". In: *BMC bioinformatics* 14.1 (2013), p. 1.

[79] Qin Ma et al. "DMINDA: an integrated web server for DNA motif identification and analyses". In: *Nucleic Acids Research* 42.W1 (2014), W12–W19. DOI: 10.1093/nar/gku315. eprint: /oup/backfile/content_public/journal/nar/42/w1/10.1093_nar_gku315/2/gku315.pdf. URL: http://dx.doi.org/10.1093/nar/gku315.

[80] Svetlin A Manavski and Giorgio Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment". In: *BMC bioinformatics* 9.2 (2008), S10.

[81] Marcel Martinez-Porchas and Francisco Vargas-Albores. "Microbial metagenomics in aquaculture: a potential tool for a deeper insight into the activity". In: *Reviews in Aquaculture* (2015).

[82] Vea Matys et al. "TRANSFAC: transcriptional regulation, from patterns to profiles". In: *Nucleic Acids Research* 31.1 (2003), pp. 374–8. ISSN: 1362-4962 (Electronic) 0305-1048 (Linking). URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=12520026.

[83] Brian J McGill et al. "Species abundance distributions: moving beyond single prediction theories to integration within an ecological framework". In: *Ecology letters* 10.10 (2007), pp. 995–1015.

[84] Antonio CA Meireles-Filho and Alexander Stark. "Comparative genomics of gene regulationconservation and divergence of cis-regulatory information". In: *Current opinion in genetics & development* 19.6 (2009), pp. 565–570. ISSN: 1879-0380 (Electronic) 0959-437X (Linking). DOI: S0959-437X(09)00154-3[pii]10.1016/j.gde.2009.10.006. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=19913403.

[85] K. Meusemann et al. "A phylogenomic approach to resolve the arthropod tree of life". In: *Molecular Biology and Evolution* 27.11 (2010), pp. 2451–64. ISSN: 1537-1719 (Electronic) 0737-4038 (Linking). DOI: msq130[pii]10.1093/molbev/msq130. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=20534705.

[86] Huaiyu Mi et al. "PANTHER version 10: expanded protein families and functions, and analysis tools". In: *Nucleic acids research* 44.D1 (2015), pp. D336–D342.

[87] Eugene W. Myers and Webb Miller. "Optimal alignments in linear space". In: *Bioinformatics* 4.1 (1988), p. 11. DOI: 10.1093/bioinformatics/4.1.11. eprint: /oup/backfile/Content_public/Journal/bioinformatics/4/1/10.1093/bioinformatics/4.1.11/2/4-1-11.pdf. URL: +%20http://dx.doi.org/10.1093/bioinformatics/4.1.11.

[88] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: http://dx.doi.org/10.1016/0022-2836(70)90057-4. URL: http://www.sciencedirect.com/science/article/pii/0022283670900574.

[89] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[90] M. Nei and T. Gojobori. "Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions". In: *Molecular Biology and Evolution* 3.5 (1986), pp. 418–426.

[91] Patrick Ng. "dna2vec: Consistent vector representations of variable-length k-mers". In: *arXiv preprint arXiv:1701.06279* (2017).

[92] Patrick Ng and Uri Keich. "GIMSAN: a Gibbs motif finder with significance analysis". In: *Bioinformatics* 24.19 (2008), p. 2256. DOI: 10.1093/bioinformatics/btn408. eprint: /oup/backfile/Content_public/Journal/bioinformatics/24/19/10.1093/bioinformatics/btn408/2/btn408.pdf. URL: +%20http://dx.doi.org/10.1093/bioinformatics/btn408.

[93] John Nickolls et al. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53.

[94] Marco S Nobile et al. "Graphics processing units in bioinformatics, computational biology and systems biology". In: *Briefings in bioinformatics* (2016), bbw058.

[95] Vera van Noort, Berend Snel, and Martijn A Huynen. "Predicting gene function by conserved co-expression". In: *TRENDS in Genetics* 19.5 (2003), pp. 238–242.

[96] NVIDIA. *GPU-Accelerated Applications for HPC Industries*. Web Page. 2013. URL: http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf.

[97] CUDA Nvidia. "Compute unified device architecture programming guide". In: (2007).

[98] D. T. Odom et al. "Tissue-specific transcriptional regulation has diverged significantly between human and mouse". In: *Nat Genet* 39.6 (2007), pp. 730–2. ISSN: 1061-4036 (Print) 1061-4036 (Linking). DOI: ng2047[pii]10.1038/ng2047. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=17529977.

[99] Uwe Ohler and Heinrich Niemann. "Identification and analysis of eukaryotic promoters: recent computational approaches". In: *Trends in Genetics* 17.2 (Feb. 2001), p. 5660. ISSN: 0168-9525. DOI: 10.1016/S0168-9525(00)02174-0.

[100] Gary J. Olsen et al. "The Ribosomal Database Project". In: *Nucleic Acids Research* 20.suppl (1992), p. 2199. DOI: 10.1093/nar/20.suppl.2199. eprint: /oup/backfile/Content_public/Journal/nar/20/suppl/10.1093_nar_20.suppl.2199/2/20-suppl-2199.pdf. URL: +%20http://dx.doi.org/10.1093/nar/20.suppl.2199.

[101] N. R. Pace. "Mapping the tree of life: progress and prospects". In: *Microbiol Mol Biol Rev* 73.4 (2009), pp. 565–76. ISSN: 1098-5557 (Electronic) 1092-2172 (Linking). DOI: 73/4/565[pii]10.1128/MMBR.00033-09. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=19946133.

[102] B. Pang et al. "Accelerating large-scale protein structure alignments with graphics processing units". In: *BMC Res Notes* 5 (2012), p. 116. ISSN: 1756-0500 (Electronic) 1756-0500 (Linking). DOI: 1756-0500-5-116[pii]10.1186/1756-0500-5-116. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=22357132.

[103] Nikitas Papangelopoulos et al. "State-of-the-art GPGPU applications in bioinformatics". In: *International Journal of Systems Biology and Biomedical Technologies (IJSBBT)* 2.4 (2013), pp. 24–48.

[104] L. W. Parfrey et al. "Broadly sampled multigene analyses yield a well-resolved eukaryotic tree of life". In: *Syst Biol* 59.5 (2010), pp. 518–33. ISSN: 1076-836X (Electronic) 1063-5157 (Linking). DOI: syq037[pii]10.1093/sysbio/syq037. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=20656852.

[105] Giulio Pavesi et al. "Weeder Web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes". In: *Nucleic Acids Research* 32.suppl$_2$ (2004), W199–W203. DOI: 10.1093/nar/gkh465. eprint: /oup/backfile/content_public/journal/nar/32/suppl_2/10.1093/nar/gkh465/3/gkh465.pdf. URL: +%20http://dx.doi.org/10.1093/nar/gkh465.

[106] W. R. Pearson and D. J. Lipman. "Improved tools for biological sequence comparison". In: *Proc Natl Acad Sci U S A* 85.8 (1988), pp. 2444–8. ISSN: 0027-8424 (Print) 0027-8424 (Linking). URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=3162770.

[107] M. Pop. "Genome assembly reborn: recent computational challenges". In: *Brief Bioinform* 10.4 (2009), pp. 354–66. ISSN: 1477-4054 (Electronic) 1467-5463 (Linking). DOI: bbp026[pii]10.1093/bib/bbp026. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=19482960.

[108] Amy J Powell et al. "Altered patterns of gene duplication and differential gene gain and loss in fungal pathogens". In: *BMC genomics* 9.1 (2008), p. 147.

[109] Junjie Qin et al. "A human gut microbial gene catalogue established by metagenomic sequencing". In: *nature* 464.7285 (2010), pp. 59–65.

[110] J. Reneker et al. "Long identical multispecies elements in plant and animal genomes". In: *Proc Natl Acad Sci U S A* 109.19 (2012), E1183–91. ISSN: 1091-6490 (Electronic) 0027-8424 (Linking). DOI: 1121356109[pii]10.1073/pnas.1121356109. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=22496592.

[111] Patricia Renesto et al. "Genome-based design of a cell-free culture medium for¡ i¿ Tropheryma whipplei¡/i¿". In: *The Lancet* 362.9382 (2003), pp. 447–449.

[112] German Retamosa et al. "Prefiltering Model for Homology Detection Algorithms on GPU". In: *Evolutionary Bioinformatics Online* 12 (2016), p. 313.

[113] D.A. Robinson, E.J. Feil, and D. Falush. *Bacterial Population Genetics in Infectious Disease*. Wiley, 2010. ISBN: 9780470600115. URL: https://books.google.com/books?id=MPWJIcTxcyQC.

[114] Indranil Roy and Srinivas Aluru. "Finding motifs in biological sequences using the micron automata processor". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, pp. 415–424. ISBN: 1479937991.

[115] Jason Sanders and Edward Jabdrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[116] Nadathur Satish et al. "Can traditional programming bridge the Ninja performance gap for parallel computing applications?" In: Portland, Oregon: IEEE Comp. Society, 2012, pp. 440–451.

[117] Dwayne C Savage. "Microbial ecology of the gastrointestinal tract". In: *Annual Reviews in Microbiology* 31.1 (1977), pp. 107–133.

[118] Michael C Schatz et al. "High-throughput sequence alignment using Graphics Processing Units". In: *BMC Bioinformatics* (2007).

[119] Patrick D Schloss and Jo Handelsman. "Introducing DOTUR, a computer program for defining operational taxonomic units and estimating species richness". In: *Applied and environmental microbiology* 71.3 (2005), pp. 1501–1506.

[120] Patrick D Schloss and Sarah L Westcott. "Assessing and improving methods used in operational taxonomic unit-based approaches for 16S rRNA gene sequence analysis". In: *Applied and environmental microbiology* 77.10 (2011), pp. 3219–3226.

[121] Bertil Schmidt. *Bioinformatics: high performance parallel computer architectures*. CRC Press, 2010.

[122] D. Schmidt et al. "Five-vertebrate ChIP-seq reveals the evolutionary dynamics of transcription factor binding". In: *Science* 328.5981 (2010), pp. 1036–40. ISSN: 1095-9203 (Electronic) 0036-8075 (Linking). DOI: science.1186176[pii] 10.1126/science.1186176. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=20378774.

[123] Klaus U Schulz and Stoyan Mihov. "Fast string correction with Levenshtein automata". In: *International Journal on Document Analysis and Recognition* 5.1 (2002), pp. 67–85. ISSN: 1433-2833.

[124] Claude Elwood Shannon. "A mathematical theory of communication". In: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), pp. 3–55.

[125] Jay Shendure and Hanlee Ji. "Next-generation DNA sequencing". In: *Nature Biotechnology* 26.10 (2008), pp. 1135–45. ISSN: 1546-1696 (Electronic). DOI: nbt1486[pii]10.1038/nbt1486. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=18846087.

[126] Edward H Simpson. "Measurement of diversity." In: *Nature* (1949).

[127] Saurabh Sinha and Martin Tompa. "Discovery of novel transcription factor binding sites by statistical overrepresentation". In: *Nucleic acids research* 30.24 (2002), pp. 5549–5560.

[128] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: http://dx.doi.org/10.1016/0022-2836(81)90087-5. URL: http://www.sciencedirect.com/science/article/pii/0022283681900875.

[129] TF Smith and MS Waterman. "Identification of common molecular subsequences. J. Mol. Biol". In: (1981).

[130] Peter HA Sneath, Robert R Sokal, et al. *Numerical taxonomy. The principles and practice of numerical classification.* 1973.

[131] Robert R Sokal. "Phenetic taxonomy: theory and methods". In: *Annual Review of Ecology and Systematics* (1986), pp. 423–442.

[132] EaBMG Stackebrandt and BM Goebel. "Taxonomic note: a place for DNA-DNA reassociation and 16S rRNA sequence analysis in the present species definition in bacteriology". In: *International Journal of Systematic Bacteriology* 44.4 (1994), pp. 846–849.

[133] James T Staley. "Biodiversity: are microbial species threatened?: Commentary". In: *Current Opinion in Biotechnology* 8.3 (1997), pp. 340–345.

[134] John E Stone et al. "Accelerating molecular modeling applications with graphics processors". In: *Journal of computational chemistry* 28.16 (2007), pp. 2618–2640.

[135] John E Stone et al. “Atomic detail visualization of photosynthetic membranes with GPU-accelerated ray tracing”. In: *Parallel computing* 55 (2016), pp. 17–27.

[136] Garret Suen et al. “An insect herbivore microbiome with high plant biomass-degrading capacity”. In: *PLoS genetics* 6.9 (2010), e1001129.

[137] Yijun Sun et al. “A large-scale benchmark study of existing algorithms for taxonomy-independent microbial community analysis”. In: *Briefings in bioinformatics* (2011), bbr009.

[138] Adam Szalkowski et al. “SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2”. In: *BMC Res Notes* 1 (2008). DOI: 10.1186/1756-0500-1-107.

[139] David Tarditi, Sidd Puri, and Jose Oglesby. “Accelerator: using data parallelism to program GPUs for general-purpose uses”. In: *SIGARCH Comput. Archit. News* 34.5 (2006), pp. 325–335. ISSN: 0163-5964. DOI: 10.1145/1168919.1168898.

[140] Sarah A Teichmann and M Madan Babu. “Conservation of gene co-regulation in prokaryotes and eukaryotes”. In: *TRENDS in Biotechnology* 20.10 (2002), pp. 407–410.

[141] Dawn Thompson, Aviv Regev, and Sushmita Roy. “Comparative analysis of gene regulatory networks: from network reconstruction to evolution”. In: *Annual review of cell and developmental biology* 31 (2015), pp. 399–428. ISSN: 1081-0706.

[142] J. D. Thompson, D. G. Higgins, and T. J. Gibson. “CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice.” In: *Nucleic Acids Research* 22 (1994), pp. 4673–4680.

[143] Andrew Todd et al. “Parallel Gene Upstream Comparison via Multi-Level Hash Tables on GPU”. In: *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*. IEEE. 2016, pp. 1049–1058.

[144] Martin Tompa et al. “Assessing computational tools for the discovery of transcription factor binding sites”. In: *Nature biotechnology* 23.1 (2005), p. 137.

[145] Susannah Green Tringe and Edward M Rubin. “Metagenomics: DNA sequencing of environmental samples”. In: *Nature reviews genetics* 6.11 (2005), pp. 805–814.

[146] Huan Truong et al. “Large-scale pairwise alignments on GPU clusters: exploring the implementation space”. In: *Journal of Signal Processing Systems* 77.1-2 (2014), pp. 131–149.

[147] Peter J Turnbaugh et al. “An obesity-associated gut microbiome with increased capacity for energy harvest”. In: *Nature* 444.7122 (2006), pp. 1027–131.

[148] Jacques Van Helden et al. “Application of regulatory sequence analysis and metabolic network analysis to the interpretation of gene expression data”. In: *Computational Biology*. Springer, 2001, pp. 147–163.

[149] Nico M Van Straalen and Dick Roelofs. *An introduction to ecological genomics*. OUP Oxford, 2011.

[150] Effy Vayena and John Tasioulas. “The ethics of participant-led biomedical research”. In: *Nature biotechnology* 31.9 (2013), pp. 786–787.

[151] J. C. Venter et al. “Environmental genome shotgun sequencing of the Sargasso Sea”. In: *Science* 304.5667 (2004), pp. 66–74. ISSN: 1095-9203 (Electronic). DOI: 10.1126/science.10938571093857[pii]. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=15001713.

[152] Thiruvengadam Vijayaraghavan et al. “Design and Analysis of an APU for Exascale Computing”. In: ().

[153] P. D. Vouzis and N. V. Sahinidis. “GPU-BLAST: using graphics processors to accelerate protein sequence alignment”. In: *Bioinformatics* 27.2 (2010), pp. 182–8. ISSN: 1367-4811 (Electronic) 1367-4803 (Linking). DOI: btq644[pii]10.1093/bioinformatics/btq644. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=21088027.

[154] G. P. Wagner and V. J. Lynch. “The gene regulatory logic of transcription factor evolution”. In: *Trends Ecol Evol* 23.7 (2008), pp. 377–85. ISSN: 0169-5347 (Print) 0169-5347 (Linking). DOI: S0169-5347(08)00163-8[pii]10.1016/j.tree.2008.03.006. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=18501470.

[155] John Paul Walters et al. “MPI-HMMER-Boost: Distributed FPGA Acceleration”. In: *The Journal of VLSI Signla Processing Systems for Signal, Image, and Video Technology* 48.3 (2007), p. 6.

[156] Xiaoyu Wang et al. “Secondary structure information does not improve OTU assignment for partial 16s rRNA sequences”. In: *The ISME journal* 6.7 (2012), p. 1277.

[157] Benjamin Webb and Andrej Sali. “Protein Structure Modeling with MODELLER”. In: *Protein Structure Prediction*. Ed. by Daisuke Kihara. New York, NY: Springer New York, 2014, pp. 1–15. ISBN: 978-1-4939-0366-5. DOI: 10.1007/978-1-4939-0366-5_1. URL: https://doi.org/10.1007/978-1-4939-0366-5_1.

[158] Li WenHsiung et al. *Molecular evolution*. Sinauer Associates Incorporated, 1997.

[159] M. F. Whitford et al. “Phylogenetic analysis of rumen bacteria by comparative sequence analysis of cloned 16S rRNA genes”. In: *Anaerobe* 4.3 (1998), pp. 153–63. ISSN: 1075-9964 (Print) 1075-9964 (Linking). DOI: S1075-9964(98)90155-X[pii]10.1006/anae.1998.0155. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=16887636.

[160] Adrianto Wirawan et al. “CBESW: Sequence Alignment on the Playstation 3”. In: *BMC Bioinformatics* 9 (2008). DOI: 10.1186/1471-2105-9-377.

[161] Kenneth H Wolfe and Denis C Shields. “Molecular evidence for an ancient duplication of the entire yeast genome”. In: *Nature* 387.6634 (1997), p. 708.

[162] Derrick E Wood and Steven L Salzberg. “Kraken: ultrafast metagenomic sequence classification using exact alignments”. In: *Genome biology* 15.3 (2014), p. 1.

[163] Eric P Xing et al. “LOGOS: a modular Bayesian model for de novo motif detection”. In: *Journal of Bioinformatics and Computational Biology* 2.01 (2004), pp. 127–154.

[164] Kefu Xu et al. “Bit-Parallel Multiple Approximate String Matching based on GPU”. In: *Procedia Computer Science* 17 (2013), pp. 523–529. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2013.05.067. URL: http://www.sciencedirect.com/science/article/pii/S1877050913002007.

[165] Masahiro Yano et al. “CLAST: CUDA implemented large-scale alignment search tool”. In: *BMC bioinformatics* 15.1 (2014), p. 406.

VITA

Huan Truong was born in a small farming-based village with a population of around 2,000 near Hanoi, Vietnam.

Huan grew up in a family of four. His parents were both educators. He graduated from Truman State University (Kirksville, MO) with a Bachelor's degree in Computer Science in 2011. Since college, Huan has developed a special interest in novel computer architectures. Huan met Dr. Gavin Conant in 2012 and under Dr. Conant's guidance, became a researcher in bioinformatics with a focus on high-performance computing. During graduate school, he also interned at the Genome Institute of Singapore to work on FactPub, a project that aimed to open up the contents of academic papers to every citizen on the planet.

Huan believes that science and technology can open new horizons to underprivileged people and change lives around the world positively.