Dynamic Construction of Trie-Based Automata for Approximate K-mer Matching on Heterogeneous CPU-GPU Systems

A Thesis
presented to
the Faculty of the Graduate School
at the University of Missouri-Columbia

In Partial Fulfillment
Of the Requirement for the Degree
Master of Science

by
TYLER WADDINGTON
Michela Becchi, Thesis Supervisor
December 2016
The undersigned, appointed by the dean of the Graduate School, have examined the
thesis entitled

DYNAMIC CONSTRUCTION OF TRIE-BASED AUTOMATA
FOR APPROXIMATE K-MER MATCHING ON
HETEROGENEOUS CPU-GPU SYSTEMS

Presented by Tyler Waddington,
a candidate for the degree of master of science,
and hereby certify that, in their opinion, it is worthy of acceptance.

______________________________
DR. MICHELA BECCHI

______________________________
DR. GRANT SCOTT

______________________________
DR. WILLIAM HARRISON
ACKNOWLEDGEMENTS

I would like to first thank my advisor, Dr. Becchi. She has provided me with the most excellent opportunity and experience in letting me be a part of her lab and research. She has sparked my interest in High Performance Computing and allowed me to see its potential for a variety of applications as I move forward in my professional career. I am grateful and indebted for all that she has provided me.

I would next like to thank my family and friends for their everlasting support and love. I would not have been able to do this without them. They’re always there to cheer me on even when I may not be at my best and I love them very much for it.

Finally, I would like to thank Dr. Scott and Dr. Harrison for being a part of my committee. Their courses were some of the key points in my college career. They fostered my interest and development in some of the most relevant and useful areas of computing. They played an important role in pushing me to where I am today.

This work has been supported through NSF award CNS-1319748 and equipment donation from Nvidia Corporation.
## TABLE OF CONTENTS

Acknowledgements .................................................................................................ii

List of Figures ........................................................................................................... vi

List of Tables .............................................................................................................. ix

Abstract ...................................................................................................................... x

Chapter 1: Introduction .............................................................................................. 1

1.1 Introduction ............................................................................................................ 1

1.2 Contributions ....................................................................................................... 2

1.3 Thesis Organizaton ......................................................................................... 3

Chapter 2: Background .............................................................................................. 4

2.1 GPU Architecture ............................................................................................. 4

2.2 Finite Automata .............................................................................................. 7

2.3 Tries .................................................................................................................. 8

2.4 Tree Bitmap .................................................................................................... 9

2.5 Levenshtein Distance .................................................................................... 11

Chapter 3: Dynamic Construction of Trie-Based Automata ................................. 14

3.1 Related Work .................................................................................................... 14

3.2 Motivation ....................................................................................................... 15

3.3 System Design ............................................................................................... 17

3.3.1 System Overview .................................................................................... 17
3.3.2 Adaptation of Tree Bitmap ............................................ 17
3.3.3 Structure of Arrays...................................................... 20
3.3.4 Intermediate Data Structures........................................ 20
3.3.5 Input Stream............................................................... 21
3.3.6 Kernel Overview......................................................... 22
3.3.7 Exact Match Traversal Algorithm ................................. 23
3.3.8 Exact Match Kernel Optimized ................................. 29
3.3.9 Serial Dynamic Matching........................................... 31
3.3.10 Deltas ...................................................................... 32
3.3.11 Stream Diagram........................................................ 33
3.3.12 Concurrent Dynamic Matching................................. 33
3.3.13 Hybrid Tree Bitmap.................................................. 36
3.3.14 Levenshtein Automata and Adaptation to Tree Bitmap
.......................................................................................... 38
3.3.15 Removing $\varepsilon$-transitions .................................. 39
3.3.16 Levenshtein Tree Bitmap.......................................... 41
3.3.17 Levenshtein Kernels................................................. 43
3.4 Experimental Results ..................................................... 51
3.4.1 Kernel Optimizations................................................. 52
3.4.2 Intermediate Tree Structure Optimizations............... 56
LIST OF FIGURES

Figure 1: GPU Architecture........................................................................ 5
Figure 2: NFA.............................................................................................. 7
Figure 3: Suffix Trie.................................................................................... 3
Figure 4: Trie-Node Structure to Bit Mapping............................................ 10
Figure 5: Tree Bitmap Structure............................................................... 11
Figure 6: Levenshtein Distance Table.......................................................... 12
Figure 7: 4-Child Inmap ............................................................................ 18
Figure 8: Prefix-Sum Calculation............................................................... 19
Figure 9: Structure-of-Arrays vs. Array-of-Structures................................. 20
Figure 10: Chunk Overlapping................................................................... 21
Figure 11: Match Format........................................................................... 22
Figure 12: Correct Transitioning................................................................. 24
Figure 13: Internal Transitions Implementation.......................................... 25
Figure 14: External Transitions Implementation.......................................... 26
Figure 15: Naïve Exact Match Kernel Algorithm......................................... 27
Figure 16: Exact Match Shuffle Kernel....................................................... 29
Figure 17: Serial Execution Loop and Block Diagram ............................... 31
Figure 18: Serial Stream Diagram............................................................ 33
Figure 19: Concurrent Block Diagram........................................................ 34
Figure 20: Overlapping Execution with CUDA Streams ......................... 35
Figure 21: Hybrid Tree Bitmap ............................................................. 37
Figure 22: Hamming Distance Automata .............................................. 38
Figure 23: Levenshtein Automata .......................................................... 39
Figure 24: Removing $\epsilon$-Transitions Using $\epsilon$-Closures ............... 40
Figure 25: Levenshtein Trie ................................................................. 41
Figure 26: Levenshtein Tree Bitmap with Labeled Transition Types ...... 42
Figure 27: I-Transition Implementation ................................................. 44
Figure 28: Internal S-Transition Implementation .................................... 44
Figure 29: External S-Transition Implementation .................................... 45
Figure 30: Levenshtein Naïve Kernel Transition Order ......................... 46
Figure 31: New Match Format ............................................................... 47
Figure 32: Transition-Efficient Kernel Transition Order ....................... 48
Figure 33: External Transition Parallelism ............................................. 49
Figure 34: Queue Kernel Memory Access Patterns ................................ 50
Figure 35: Writing to Queue Implementation ....................................... 50
Figure 36: Exact Naïve vs. Shuffle Results ......................................... 52
Figure 37: Levenshtein Naïve vs. Transition-Efficient Results ............. 53
Figure 38: Levenshtein Transition-Efficient vs. Queue ....................... 55
Figure 39: Intermediate Data Structure Optimization .......................... 57
Figure 40: Serial Execution Streams ........................................................... 58
Figure 41: Concurrent Execution Streams .................................................. 60
LIST OF TABLES

Table 1: Hardware Configuration................................................................. 51
Dynamic Construction of Trie-Based Automata for Approximate K-mer Matching on Heterogeneous CPU-GPU Systems

Tyler Waddington
Dr. Michela Becchi, Thesis Supervisor

Abstract

In recent decades, mapping of a variety of species’ genomes has taken place. With the proliferation of advanced and specialized hardware architectures such as GPUs, the process has been greatly accelerated. GPUs may accelerate core algorithms used for k-mer matching and alignment but they only play a part in a bigger system.

The research contained in this thesis covers the development of a heterogeneous system that utilizes the CPU and GPU to provide a powerful dynamic system. The core content of this thesis describes a way to locate similar k-mers within a single stream in DNA bases. It achieves this by using the CPU to dynamically construct a trie-based automata structure while the GPU provides the k-mer matching mechanism. It will cover the implementation and optimization of GPU kernels used for k-mer matching, implementation of 2 intermediate automata structures operated on by the CPU, and a way to utilize CUDA Streams to hide latency within the system.
Chapter 1. Introduction

1.1 Introduction

The first multi-cellular organism’s DNA was sequenced in 1998. By 2001, 90 percent of the human genome had been sequenced by the Human Genome Project, which in turn, lead to a renaissance in genome sequencing. Private and public organizations took interest in the growing market. Shortly after, machines were able to sequence billions of DNA bases in a matter of days. The National Center for Biotechnology Information (NCBI) now houses over 170,000 genomes of various species. NCBI provides these genomes for free. The large accessibility of genomes resulted in increased efforts in Genome Mapping (although efforts in this area had existed before). Genome mapping is the process of discovering the location and functionality of genes and can be done by comparing them to previously discovered genes.

The process of gene mapping would be impossible without the assistance of computers. Tools such as FAST and BLAST have been created to align sequences of DNA in order to find similarities. The performance of these tools is generally guided by Moore’s Law, which states that the transistor density of CPUs should double about every 2 years. This roughly translates to the performance of processors doubling every two years. However, as higher processor clock speeds became more implausible due to power constraints, it became common for processors to have multiple cores. BLAST took advantage of this new architecture by implementing Parallel BLAST, the BLAST algorithm adapted for multi-core architectures.

In recent years, even further acceleration of bioinformatics applications have been sought after through the use of more specialized architectures such as General Purpose
GPUs (GPGPU). While CPUs aim for low latency, GPGPUs aim for high throughput. They achieve this high throughput through the use of many (thousands) cores operating in parallel. This architecture has attracted bioinformatics applications because of their need to process enormous amounts of DNA base pairs.

However, the migration of algorithms from CPU to GPU has often left the CPU underutilized. The main focus of the application moves onto the GPU while the CPU is mainly used for preprocessing and data transfers. The preprocessing phase mainly consists of the CPU initializing and preparing static data structures for the GPU to operate on. The CPU is then required to handle data transfers between the two.

This thesis provides a foundation for doing exact and approximate pattern matching in streams of DNA bases using a dynamically changing data structure. It takes the novel approach of distributing data structure compilation throughout an application’s lifetime. Thus, it is able to achieve concurrent use of the CPU-GPU architecture making it a truly heterogeneous system.

1.2 Contributions

This thesis describes the design and implementation of a heterogeneous CPU-GPU system used to do approximate sequence matching in a stream of DNA bases using dynamic data structures. The contributions of the thesis are:

- The design and implementation of algorithms for approximate and exact pattern matching in DNA base streams and the optimization of the aforementioned algorithms.
• The optimization and adaptation of CPU structures for dynamic use in a heterogeneous CPU-GPU environment.
• The design and implementation of a CPU-GPU system that allows for concurrent execution of the CPU and GPU.

1.3 Thesis Organization

The thesis is organized in the following manner. Chapter 2 covers background concepts needed to understand the contributions. It provides a basic overview of GPU architecture and intrinsic functions, the basics of Non-Deterministic Finite Automata (NFA), the Tree Bitmap data structure used for pattern matching, and Levenshtein Distance used as the approximate distance measure. Chapter 3 describes the design and implementation of the dynamic heterogeneous DNA base pattern matching system. The chapter will discuss the contributions covered in the previous section and discuss the experimental results for these contributions and compare them against their un-optimized counterparts. Finally, Chapter 4 will provide a conclusion to the thesis and summary of its contents.
Chapter 2: Background

2.1 GPU Architecture

GPUs are not a new technology. In fact, NVIDIA has been manufacturing GPUs since 1993. However, the advent of GPGPUs wasn’t introduced until much later. NVIDIA introduced the first programmable GPU with the GeForce 3 in 2001, however, only to a limited extent. Their first real GPGPU came with the introduction of the CUDA library and runtime in 2007. CUDA [3] introduced a way to program GPUs with relative ease, although it requires an individual to think in a highly parallel mindset. In addition, it requires the programmer to understand the hierarchical nature of the GPU.

As hinted above, GPUs are highly parallel machines. At the highest level, each GPU is composed of multiple processors. The processors are referred to as Streaming Multi-processors (SMP). These processors are able to act independently from one another. Figure 1 illustrates this high level view.
Each SMP is able to run many threads in parallel. This means a GPU is able to support many, many threads. The totality threads executing of the GPU are logically referred to as a grid. Grids are then separated in blocks of threads. If one wanted to know how many threads were executing, in total, on the GPU they would use the equation ThreadsPerBlock \times BlocksPerGrid. In turn, each block is assigned to an SMP. Multiple blocks can execute on a single SMP in parallel or concurrently.

An SMP contains many cores. The cores within an SMP are typically arranged in groups of 32 that executes in an SIMD fashion. This means that threads that execute on this group of 32 are from the same block and execute in lockstep. The group of 32 threads is referred to as a warp. Warps on a SMP may be from different blocks and can execute independently from one another.
The GPU memory is also hierarchical. At the top of the hierarchy is Global Memory. All threads in the grid have access to this memory space. An L2 cache is located between the SMPs and Global Memory. Next, each SMP has its own store of memory, typically 64KB referred to as Shared Memory. This memory is block-private. A block cannot access another block’s shared memory space. It should be noted that Shared Memory has very low latency and can be configured as an L1 cache. Finally, each SMP has access to a set of registers, 255 per thread in the Maxwell architecture.

The highly parallel nature of GPUs does not come without downfalls. The SIMD nature of warps causes multiple problems, of which this thesis notes 2. The first is referred to as *memory coalescing*. Threads in a warp access memory that is contiguous more efficiently. They are effectively able to grab a warp’s width of words in a single access. However, if this memory is scattered, it will take multiple accesses to retrieve the data; the more scattered the data, the more memory accesses required.

The second limitation is referred to as *warp divergence*. As mentioned above, threads in a warp must execute in parallel lockstep. Because of this, warps don’t handle branches well. If only a portion of the warp satisfies the condition of a branch, for example, that portion will continue to execute while the other portion of the warp sits idle and their respective cores are un-utilized. Only after the branch statement, do the threads merge and continue in lockstep.

GPUs provide special functionality in addition to the normal floating point and integer operations. These functions are typically referred to as intrinsic functions and are supported by hardware. The main goal of many of these functions, such as *ballot* and
shuffle, serve the purpose of synchronization and communication between threads within the warp. Ballot allows threads in a warp to vote. It takes a predicate as input, and outputs a bitmap; if the Nth thread’s predicate is true, the Nth bit in the bitmap is set. Shuffle provides functionality for one thread to retrieve a value from another threads register. A thread provides the variable it wishes to retrieve and which thread it wishes to retrieve it from. It returns the requested value.

2.2 Finite Automata

Finite Automata (FA) are machines used for pattern matching purposes [4]. A stream of symbols is sent into the machine sequentially, and the machine recognizes the string if it is defined by the FA. The set of symbols, Σ, that a given FA recognizes is known as its alphabet. The structure of a FA is graph-like in nature.

![NFA Diagram](image)

Figure 2: NFA; Recognizing Patterns t, and ac

Vertices, referred to as states, are connected by directed edges, referred to as transitions. Each of these transitions represents a symbol or set of symbols. The FA has a starting state, the state that is initially active before any symbols are streamed in. When the FA consumes a symbol, the FA is traversed from the active state over the transition
corresponding to the symbol, which in turn activates the adjacent state. If an active state
does not have a transition corresponding to the input symbol, it deactivates.

Recall that transitions are able to correspond to sets of symbols as well. This
thesis considers 2 major sets, the empty set and universal set. The empty set transition
will be referred to using the symbol, ε, or ε-transition. A transition on ε happens without
an input character being consumed. If state $S_a$ is connected to state $S_b$ by an ε-transition
and $S_a$ is active, $S_b$ is active as well. The universal set represents all symbols contained in
alphabet, $\Sigma$. A transition on this set is denoted using * and these transitions will be
traversed for every symbol in the $\Sigma$.

Figure 2 defines an NFA that recognizes patterns $t$, and $ac$. The NFA (Non-
Deterministic Finite Automata) allows for multiple states to be activated at once. The ε-
transition creates an “automatic” transition from state-2 to accepting state-3 denoted by
the concentric circles. The self-loop at state-0 makes 0 into a persistent state, always
staying activated. This allows the NFA to recognize its defined patterns despite where
they might lie in the input stream. Unless otherwise stated, all automata structures in this
thesis have a persistent initial state.

2.3 Tries

Tries, or Retrieval Trees [5], can be viewed as trees used for pattern matching.
Generally, for the purposes of this thesis, a trie is a specialized NFA. Each node in the
trie represents the sequence of symbols traversed to reach the specified node. Because of
this, they are typically used as dictionaries.
Going further, a trie can be classified as a *suffix trie* based on the way it is constructed. Given a string, the suffix trie contains all suffixes of the string. The suffix trie for *bubble*, contains *bubble*, *ubble*, *bble*, *ble*, *le*, and *e*.

Suffix tries are useful when handling k-mers. A *k-mer* is just a sub-sequence of length $k$ that is found in a larger sequence of symbols. Since the suffix trie is constructed from every possible k-mer in a sequence, a suffix trie-based NFA with a persistent initial is able to detect k-mers within any alignment of a given sequence. By “any alignment” it is meant that the first symbol in the k-mer can lie on any given symbol of the sequence.

### 2.4 Tree Bitmap

The Tree Bitmap structure of Eatherton et al. [6] is a trie-like data structure adapted for the use in routers. It uses a binary tree to recognize sequences of 1s and 0s. It is designed for use in IP lookup operations. As noted in the previous section, each node corresponds to a given sequence of binary symbols.
Unlike a typical pointer-based trie, the Tree Bitmap structure is stored contiguously in as a bitmap. Groups of 7 nodes are grouped into a single sub-structure called a trie-node. If a node’s bit has a logical value of 1, the node exists in the trie. If the node’s bit has logical value of 0, the node does not exist. This implies that if a node’s bit is not set, none of that node’s children are set either. Because a trie-node only represents 7 nodes, it can easily fit into a single byte, utilizing 7-out-of-8 bits. This bitmap is referred to as the internal-bitmap or *inmap*.

![Figure 4: Trie-Node Structure to Bit Mapping](image)

Figure 4 shows the inferred mapping from an inmap to the trie-node structure. Nodes are stored in a breadth-first fashion with the higher-depth nodes positioned more toward the most significant bit (MSB) with the MSB unused.

The Tree Bitmap structure is composed of trie-nodes the same way a regular tree is composed of nodes. Each trie-node has a maximum of 8 transitions connecting it to its children trie-nodes (2 transitions per bottom leaf node). In addition to the inmap, each trie-node also has an external-map or *exmap*; the exmap determines if a trie-node’s child exists.
Figure 5 exemplifies the Tree Bitmap structure. Trie-nodes are connected in a tree-like structure. Set bits in the exmap determine the existence of transitions to children nodes. The Tree Bitmap is stored as a contiguous array of trie-nodes (the arrows are for illustration and don’t represent pointers). This means the last trie-node of a given depth, or *level*, sits next to the first trie-node in next level.

The Tree Bitmap is still a traversable data structure, much like a pointer-based tree, however it is done differently. A supplementary array holds the offset for the first trie-node in each level. A trie-node’s child can be found by summing the offset of the next level, the population count of the trie-nodes previous sibling’s exmaps, and the exmap bit index of the trie-nodes child. This process will be described further in Chapter 3 of the thesis. For now, the reader should keep in mind that it is possible to traverse a Tree Bitmap structure.

### 2.5 Levenshtein Distance

Levenshtein Distance is an approximate distance measure between two strings of symbols proposed by Levenshtein [7] himself in 1966 and further work has been applied
since then to improve it [8-10]. It provides a value to determine how similar two strings might be by describing the minimum number of edits it takes to change one string to another. The edits are separated into 3 types: substitutions, insertions, and deletions.

Substitutions are a change from one symbol to another; for example, *bubble* and *babble* have an edit distance of 1 by a substitution of *u* to *a*. Insertions and deletions account for difference in length between 2 strings; *babble* and *blabble* have an edit distance of one with an insertion or deletion of *l*.

<table>
<thead>
<tr>
<th></th>
<th>#</th>
<th>G</th>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6: Levenshtein Distance Table

The rules of calculating the table are simple. An empty cell is calculated using the cells to the left, above, and left-above diagonal. If given the table represented as a matrix, the current cell under calculation is \(matrix[r][c]\), above is \(matrix[r-1][c]\), left is \(matrix[r][c-1]\), and diagonal is \(matrix[r-1][c-1]\). The value of the cell can be calculated using the following equation.
\[
\min \begin{cases} 
\text{matrix}[r-1][c]+1 \\
\text{matrix}[r][c-1]+1 \\
\text{matrix}[r-1][c-1]+0; \text{if string1}[r]=\text{string2}[c] \\
1; \text{if string1}[r]\neq\text{string2}[c]
\end{cases}
\]

string1 is lined along the rows, and string2 is lined along the columns. From string1’s point of view, insertions happen in the vertical direction (letters are added or inserted into string2 in this direction). Deletions happen in the horizontal direction, and substitutions occur in the diagonal direction. However, a substitution only occurs if the current letters of corresponding letters of string1 and string2 are not equal, hence the “\(\neq\)” condition in the above equation. Although, this thesis’ implementation does not directly use this approach, it may be helpful in understanding ideas covered in Chapter 3.
Chapter 3: Dynamic Construction of Trie-Based Automata

3.1 Related Work

Dynamic construction of automata is an area that has been sparsely researched. However, there has been research toward the implementation of NFA on GPU architectures; Cascarano et al. introduced an approach to GPU NFA representation [11] called iNFAnt. Similarly CSR graph adaptations have been implemented [13]. These projects lean toward a static implementation of FA and are general purpose. They have the added benefit of allowing the user to implement (theoretically) automata of any nature. However, their limitations lie in the realm of the number of FA states that can be represented. iNFAnt for example, is able to represent 65,536 states. In addition, the complexity of such project is typically dependent on the number of states however [12] has taken steps to mitigate this limitation.

Research has been carried out on the utilization of GPUs to compute approximate distance calculations such as Levenshtein Distance: [dynamic programming, bit parallel]. The popular method of doing these calculations is dynamic programming [15,17]. A lot of effort has gone into this area in the recent decades for use in sequence alignment. These methods are efficient for use in sequence-to-sequence alignment. However, they are not suitable for use in stream-like applications or comparison against a large number of possible. Algorithms introduced in [18,19] use Myer’s bit parallel algorithm [16] and come closer to the nature of this thesis implementation by using bitmaps to represent dynamic programming tables however they still rely on dynamic programming techniques. Steve Hanov proposed a trie-based
structure together with dynamic programming [14] to allow comparison against a larger
dictionary, but its recursive nature is ill suited for the GPU. Trie’s have widely been used
for exact distance matching. Later sections of this thesis will discuss their unique
adaption toward approximate distance matching.

Other architectures have been used to implement automata such as FPGAs and
the Micron Automata Processor [20,22]. Such implementations have the benefit of
representing states using registers; a register holding a logical ‘1’ represents an active
state. Transitions are represented using a physical wire and a comparator. Use of digital
logic in this manner, means a transition from one state to another can occur in a single
clock cycle, allowing millions of symbols to be processed every second. However,
compilation phases for these architectures typically require time in the magnitude of
hours, making their use cases static. IBM introduced a regular expression co-processor
[21], which may be a happy medium between ultra-fast architecture and fast compile
times. It is more tailored toward regular expression matching than $k$-mer matching
though.

3.2 Motivation

The current state-of-the-art implementations provide efficient solutions to
approximate distance matching on GPU and separately, the execution of FA on
GPU. Additionally, specialized architectures provide high performance in FA
execution. These applications have limitations in the following areas. The calculation of
approximate distances typically uses dynamic programming techniques and is a poor
choice for streaming applications. FA on GPUs typically have a large memory footprint
and thus there is a limit to the number of representable states. Specialized architectures require a large amount of time to compile FA to a format representable by the architecture. This thesis looks at a means to mitigate these limitations.

*Streaming limitations due to Dynamic Programming* – Dynamic programming techniques for approximate distance matching consist of comparing two k-mers. When working with a stream, this requires looking at every possible k-mer in a stream and comparing it against every k-mer in a dictionary. This equates to \((L - k + 1) \times D\) k-mer comparisons, where \(L\) is the length of the stream, \(k\) is the length of the k-mer, and \(D\) is the number of k-mers in the dictionary. This complexity makes dynamic programming techniques an unfriendly application for streaming. The thesis provides a stream friendly implementation using specialized automata for approximate distance matching.

*FA state limitations due to large footprint data structures* – General automata structures can be represented in a number of ways. A typical way to represent them is using a list of transition as source-destination pairs such that every state is enumerated. A supplementary bitmap is used to determine active states. For large automata, structures can be very large if there are many transitions. The thesis mitigates eliminates the need for represented transitions by using an automata specialized toward the application space. Transitions are inferred from the data structure itself.

*Limited support for dynamic automata* – Thus far, automata based projects have remained static; automata are built and then reside on a coprocessor. Typically, co-
processors require large amounts of time to compile for, making dynamic growth of automata on these architectures nigh impossible. This leaves a gap that can be filled using architectures that are relatively easily to compile for. The thesis provides data structures that can be compiled fairly quickly for use on the GPU.

3.3 System Design

3.3.1 System Overview

The system is heterogeneous and utilizes both the CPU and GPU. The CPU is responsible for building the NFA (also referred to as trie) and the GPU is responsible for evaluating, or traversing the NFA given an input stream. First, the CPU generates a trie from a small portion of the input stream; this is referred to as seeding the trie. The reasoning for this will be discussed in a later section. The trie is only an intermediate data structure; it is converted to the Tree Bitmap structure and then sent to the GPU. The GPU evaluates the NFA against a portion or chunk of the input stream and then tells the CPU what changes need to be made to the trie structure. The CPU makes these changes, converts the new trie to the Tree Bitmap and the process continues until the entire stream has been processed.

3.3.2 Adaptation of Tree Bitmap

The Tree Bitmap structure is a prime candidate for use in base-pair comparisons. It is targeted toward alphabets of small sizes. The benefit is that a single trie-node’s inmap or exmap fits nicely within a computer word. The original Tree Bitmap structure
natively supports a binary alphabet. However, it is adapted to an alphabet of 4 symbols for this thesis.

![Figure 7: 4-Child Inmap (Exmap Excluded)](image)

The inmap requires 21 bits to be represented properly: 1 bit for the root, 4 for its children, and 16 for its grandchildren. The 4-symbol inmap fits within a 32-bit word. Each of the 16 grandchildren can then have 4 exiting transitions each or 64 exiting transitions in total. Thus, the exmap can fit within a 64-bit word.

Tree Bitmap requires a calculation to be done for a transition from a trie-node to one of its children.

\[
\text{levelOffset} + \text{previousNiblings} + \text{popc}((1 \ll \text{child})-1 \& \text{exmap}) \tag{1} \\
\text{prefixSum} + \text{popc}((1 \ll \text{child})-1 \& \text{exmap}) \tag{2}
\]

Here \text{levelOffset} is the provided offset of the level that the trie-node resides in. \text{previousNiblings} is the number of total number of children possessed by all of the previous trie-node’s siblings in this level. For a given trie-node with index \( tn \), the \text{previousNiblings} can be calculated as shown in Figure 8.
for( i=levelOffset; i<tn; i++):
    previousNiblings += popc( Tree Bitmap[i].exmap )

Figure 8: Prefix-Sum Calculation

Here, previousNiblings is initialized to 0. Back in Equation 1, popc() calculates the number of set bits in a given word. A mask is calculated using \((1<<\text{child})-1\) to mask only the children before the desired child and a bitwise-and (\&) with exmap ensures these are the only children seen when counting them using popc.

The astute reader will notice that it is cumbersome to calculate previousNiblings for every traversal. For this reason, a new member to the trie-node is added called the prefix-sum. This value is equal to levelOffset + previousNblings and is calculated whenever the Tree Bitmap structure is created. As a result, a simple lookup takes the place of the calculation in Figure 8. The new equation is presented by Equation 2.

In this system, the GPU holds a Tree Bitmap structure used for querying and an active bitmap. The active bitmap is used for the traversal of the Tree Bitmap to represent active states and is composed of active inmaps. How it is actually used will be discussed in the following sections covering traversal algorithms. When used in the context of traversal, the active inmaps may be referred to as inmaps for brevity.

Finally, the strength of representing the trie-based NFA as a bitmap should be noted. As stated above, it is common to represent transitions as a 32-bit source-destination pair (16-bits for source and 16-bits for destination). The provided bitmap representation encodes 20 transitions in a 32-bit inmap and 64 transitions in a 64-bit exmap. This equates to an average of 28 transitions per 32-bit word. In addition,
transitions are localized to the given trie-node, meaning that transitions from one state to another can be executed within the same computer word.

3.3.3 Structure of Arrays

The first notion when understanding the Tree Bitmap structure is to view it as an array of trie-nodes. Each trie-node can then be though of as a struct.

```c
struct TrieNode{
    inmap;
    exmap;
    prefix_sum;
    active_inmap;
}
struct Tree Bitmap{
    inmaps[N];
    exmaps[N];
    prefix_sums[N];
    active_inmaps[N];
}
```

Figure 9: Structure-of-Array (SoA) vs. Array-of-Structures (AoS)

This representation of a Tree Bitmap may be efficient for the CPU depending on the application. However, in this thesis, the Tree Bitmap is to be evaluated on the GPU. Arrays-of-Structures such as the one depicted in Figure 9 are inefficient on GPUs given that GPUs prefer coalesced memory access. Instead the structure is organized as a Structure-of-Arrays. The Structure-of-Arrays allows for more coalesced memory accesses on the GPU. It should be noted that AoS is used to illustrate code in this thesis for simplicity. In reality, all GPU code discussed here-in utilizes SoA.

3.3.4 Intermediate Data Structure

Recall that the main goal of the thesis is to create dynamically growing automata. While Tree Bitmap is a relatively efficient trie-based structure for the GPU (given its
contiguous nature), it is a poor choice as a dynamic structure. Frequent edits to the Tree Bitmap would incur large penalties due to memory reallocations.

Instead, pointer based structures work well as dynamically growing structures. A generic pointer-based tree structure is used. Each node in the tree can have a maximum of 4 children (one for each base symbol). Such a structure is easily updated and can be converted to a Tree Bitmap structure with relatively good performance.

3.3.5 Input Stream

The input stream is initially loaded onto the GPU. Rather than loading it piece-by-piece whenever the GPU needs it, it is loaded in its entirety. This avoids any additional overhead accumulated from multiple data transfers to the GPU. When a chunk from the stream is needed by the GPU, the boundaries of the chunk are sent so that the executing kernel knows which part of the stream to use.

![Figure 10: Chunk Overlapping; For k=4](image)

The chunks do not have clean boundaries; the boundaries of each chunk must be overlapped with adjacent chunks to avoid missing matches. For example, if a k-mer lies on the boundary between 2 non-overlapped chunks, a portion of it would belong to the
left chunk and a portion to the right chunk, and the matching kernel would receive the k-
mer’s constituents but never the whole k-mer. This can be avoided by overlapping the
chunks by $k-1$ symbols as seen in Figure 10.

### 3.3.6 Kernel Overview

The main goal of the GPU kernels is to evaluate the NFA against an input stream; this means making transitions on active states for each symbol of the input stream. In addition, the kernel should report when a sequence in the input stream matches a sequence defined by the Tree Bitmap structure. These matches are written to a buffer in global memory referred to as the *match buffer*. Matches are a 64-bit word and have the following format in Figure 11.

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>32</th>
<th>31</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Stream Index</td>
<td>Trie-Node</td>
<td>Depth</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 11: Match Format

The value of type-bit is ‘0’ for matches and is discussed in the next paragraph. The stream index is the index into the stream at which the match occurred. The trie-node is the index of the trie-node in the bitmap at which the match occurred. The depth is the depth at which the match happened; with exact matches, this should always be at depth k.

At no point does the GPU edit the Tree Bitmap structure, except for active inmaps. As mentioned above, only the CPU edits the intermediate tree structure. But the CPU requires some hint to know what part of the intermediate structure it needs to edit; it
needs to know which k-mers have been found in the stream that do not already exist in the trie. The kernel provides this hint in the form of mismatches. Mismatches have the same format as defined in Figure 11. The type-bit is set ‘1’ to differentiate it from a match. The stream index is the index in the stream at which the traversal algorithm found there was no transition for the given symbol. The depth is the depth within the tree at which the mismatch happened; depth may have a value in the range [1,k] where k is the length of the k-mer. Every time the traversal algorithm finds a pattern in the stream that does not exist in the trie, a mismatch is generated and pushed into the match buffer.

3.3.7 Exact Match Traversal Algorithm

The traversal algorithm steps through the input stream and makes transitions on each of the active states in the active bitmap. For now, this means iterating over each trie-node and transitioning each one that contains an active state. This exhaustive search for active trie-nodes is inefficient and will be discussed in a later section.

Dependencies between states of different depths require that transitions be done starting with larger depths. For example, transitions must be done first from depth $k-1$ to $k$, followed by $k-2$ to $k-1$, $k-3$ to $k-2$, and ending with 0 to 1.
Figure 12: Correct Transitioning (a) Initial State (b) Deepest Transition First(Correct) (c) Shallowest Transition First(Incorrect)

Figure 12 illustrates the reasoning for the above statement. If iterating from smallest-to-largest depth, a false transition is made. This problem is avoided when iterating from largest-to-smallest depth.

Given the above, the kernel iterates from the deepest level to shallowest level in the Tree Bitmap. Trie-nodes within a given level can be processed in any given order seeing that there are no dependencies between trie-nodes in the same level.

Two different transitions exist when dealing with the Tree Bitmap structure: internal transitions and external transitions. Internal transitions are transitions that occur within the inmap. External transitions occur between a trie-node in level $l$ to another trie-node in level $l+1$ using the trie-node a level $l$’s exmap.
The first step of transitioning an inmap is to isolate the depths of the inmap. To reiterate, depth_1 must be transitioned before depth_0. depth_2 is not transitioned, as depth_2’s transitions are external transitions. When transitioning a given depth, the function generate_next_depth() is called. generate_next_depth() looks at each set bit at the given depth, multiplies its index by 4 to indicate the group of 4 children the transition belongs to, and adds \(tx\) (an enumeration of a,c,t, or g) to specify the offset within the group of children.

The generated depths are then shifted to their respected position in the inmap and ORed to create a complete inmap.
In Figure 14, external transitions are handled in a similar manner to internal transitions. `generate_next_depth()` is used to determine which transitions are taken from the `depth_2` of a given inmap. The result is a mask that is ANDed with the trie-nodes exmap to determine transitions that are to be taken and necessarily exist; it is referred to as the `exmap mask`. Each set bit in the exmap mask is iterated over, and using Equation 2, the trie-node on the other end of the transition is ORed with 0x1 to activate its root node. An important note: the `exmap mask` should be generated before the inmap is transitioned as to avoid the false activations of Figure 12.

```c
1. exmap_transitions( trieNode, tx ):
2.   exmap_mask = generate_next_depth( trieNode.act_inmap >> 5 )
3.   return trieNode.exmap & (exmap_mask << tx)
```
Figure 15: Naïve Exact Match Kernel Algorithm
Figure 15 illustrates the traversal kernel. The block size is assumed to be 2 for simplicity. Starting with Figure 15a, the threads access the trie-nodes in an interleaved fashion and transition each trie-node’s inmap on each visit. They then synchronize and continue with level 1 in Figure 15b. They execute the external transitions first (bold arrows) using the trie-node’s exmap followed by inmap transitions. Finally, the top trie-node is transitioned in Figure 15c. At this $t_1$ is idle while thread $t_0$ executes.

A match occurs if $\text{AND}$ing the queried inmap with the active inmap returns greater than zero and the depth of an active state equals $k$. Mismatches occur when $\text{AND}$ing active inmaps with the complemented queried inmap; a returned value greater than zero determines that a state is active where a state does not already exist. In this case, the mismatch is recorded and these falsely active states are cleared by $\text{AND}$ing the active inmap with the queried inmap.

The traversal algorithm is divided into block level parallelism and thread level parallelism. The block level parallelism assigns each block a piece of the input stream chunk. If there are 24 blocks, the stream chunk is divided into 24 pieces. Each block executes the algorithm discussed in the previous section.

The thread parallelism describes how threads are parallelized within a block. All threads within a block must be processing the same stream symbol and level at a given time step. Not doing so breaks dependencies and results in undefined behavior. Threads in a block process all of the trie-nodes in a level in parallel and are synchronized via CUDA’s `__syncthreads()` function on each change in level.
3.3.8 Exact Match Kernel Optimized

The original traversal kernel naively iterates over every trie-node in the Tree Bitmap structure. This strategy is inefficient because the bitmap may have a large degree of inactive states. This is the case for exact matching.

The transitions leaving a state in a trie-based NFA are mutually exclusive; only one transition can be taken for a given input symbol $a$, $c$, $t$, or $g$. If the symbol $a$ is received, only $a$-transitions may be taken. This implies that an active state can only spawn 1 active state at the next depth. Because the root of a trie is the only state at depth 0, only one state can be activated at depth 1. Following this train of logic shows that there can only be $k+1$ (one per depth) active states in the trie at any point in time.

A new algorithm is implemented to take advantage of the above observation. Each thread is assigned to a depth in the Tree Bitmap structure. On each input symbol, a thread transitions their trie-node and passes it on to the next thread. Every 3rd thread is responsible for executing external transitions from one trie-node to another. The $k$th thread is responsible for determining whether a match exists. All threads are responsible for reporting mismatches.

![Diagram](image)

Figure 16: Exact Match Shuffle Kernel
Figure 16 illustrates the flow of data from one thread to another. Parallelism is extracted at the warp and thread level. Different warps execute on different pieces of the input stream. Each thread in the warp is assigned a lane_id, its identifier in the given warp. There is a direct mapping from a thread’s assigned depth to its lane_id; a thread of depth \( i \) has a lane_id of \( i \). Thread \( t_0 \) holds the root trie-node (whose root bit is always 1); it transitions the trie-node and passes it to thread \( t_1 \) via CUDA’s intrinsic function \texttt{__shfl()}_. It should be noted that all threads are transitioning their trie-nodes in parallel, however it is easier to explain it in a serial manner. On the next symbol, \( t_1 \) transitions the trie-node received from \( t_0 \) and passes it to \( t_2 \). On the next symbol, \( t_2 \) calculates the external transition using Equation 2, reads this trie-node from memory, and passes it to \( t_3 \). This process continues down the chain of threads.

As stated before, every thread is responsible for detecting mismatches and the \( k^{th} \) thread detects matches. After executing, transitions and \texttt{__shfl()}s for a given symbol, all matches and mismatches are reported to the match buffer. Using the intrinsic function \texttt{__ballot()}_, threads vote on who has mismatches or matches. If any of them do, they use the ballot returned from the function to write contiguously, in a coalesced fashion, to the match buffer.

Utilizing the warp in this way can be detrimental to the occupancy of the GPUs hardware. For now, an improvement in this area remains for future iterations of the project but an idea to mitigate the problem is provided anyway. If \( k=5 \), for instance, only 6 \((k+1)\) GPU cores are being utilized out of the 32 provided for the warp. Luckily, the \texttt{__shfl()} function allows the user to treat a warp size as any power of 2 up to 32. This
means with $k=5$ we can use 4 “warp” sizes of 8, thus theoretically achieving 4 times the occupancy. However, if $k=16$, the remaining 15 cores will remain unutilized.

### 3.3.9 Serial Dynamic Matching

The entirety of the input stream is initially loaded onto the GPU. A portion of the input stream is used as a seed for the intermediate data structure or trie; the trie is initialized using all existing k-mers in the seed. A non-seeded trie would cause a mismatch for every input symbol leading to unnecessary overhead. After seeding, the process enters the execution loop.

```plaintext
1. trie.seed(stream, seed_size)
2. while (stream.chunksRemaining()) {
   3.   chunk = stream.chunkify()
   4.   treeBitmap = trie.toBitmap()
   5.   treeBitmap.toGPU()
   6.   chunk.toGPU()
   7.   kernel.launch(chunk, treeBitmap, matchBuffer)
   8.   matchBuffer.toHost()
   9.   deltas(trie, matchBuffer)
```

![Figure 17: Serial Execution Loop and Block Diagram](image-url)

Figure 17: Serial Execution Loop and Block Diagram
Figure 17 features the execution loop and block diagram of the modules that make up the system. The first step in the execution loop is to call chunkify() which grabs a chunk from the input stream. The intermediate tree is then converted to the Tree Bitmap structure via the toBitmap() method. From here, the Tree Bitmap structure and chunk are loaded onto the GPU and the kernel is launched.

During execution the kernel is recording matches and mismatches into the match buffer. The match buffer is partitioned into a host-side and device-side buffer. After kernel execution, the device-side buffer is transferred to the host-side buffer via the toHost() method. Proceeding the transfer of the match buffer, the process enters the deltas() function which adds mismatched k-mers to the intermediate tree.

3.3.10 Deltas

Deltas represent changes that are made to the intermediate trie. The system uses information supplied by the mismatches to apply the changes. Specifically, the depth, stream index, and k are used in locating the k-mer for which the mismatch occurred. The indices for the first and last characters are calculated as follows.

\[
\begin{align*}
\text{streamIndex} - \text{depth} + 1, \\
\text{streamIndex} + (k - \text{depth}) 
\end{align*}
\]  

(3)

The two indices, represented by Equation 3, are used to specify the k-mer to be added to the trie. If the k-mer exists in the trie already, it had been added at a previous point in the current set of mismatches; in this case the mismatch is converted to a match.
3.3.11 Stream Diagram

Finally, the process of calculating mismatches and adding deltas to the trie can be represented in a more succinct way using a time diagram shown in Figure 18. This notation will help with optimization of the algorithm in the following section.

```
| chunkify() | toBitmap() | HtoD | Kernel | DtoH | deltas() |
```

Figure 18: Serial Stream Diagram

The diagram illustrates the order of events as time progresses, where each cell represents an event. \textit{HtoD} and \textit{DtoH} represent a transfer either from host-to-device or device-to-host, respectively. The kernel executes between the two transfers and cannot start until the \textit{HtoD} transfer completes. \textit{DtoH} transfers are synchronized with the device using \texttt{cudaDeviceSynchronize()}, as noted by the bold line between the kernel and \textit{DtoH} cell. This means \textit{DtoH}s do not begin until the kernel returns.

3.3.12 Concurrent Dynamic Matching

The concurrent version takes a similar approach as the serial approach, however many of the modules in the serial approach must be duplicated as seen in Figure 19.
To allow for concurrency, there must now be 2 chunks and 2 match buffers. This allows for one of the modules to be used on the GPU while the other is being operated on the CPU. More specifically, one match buffer can be filled with matches by the GPU, while the CPU is computing deltas from the other buffer. On each loop of the execution, the two are multiplexed, switching between one set of buffer/chunk and another.

Concurrent dynamic matching attempts to make the matching process more efficient by overlapping CPU and GPU execution. Another row is introduced to the diagram in Figure 18. Overlapping execution is denoted by actions placed in the same column.
Figure 20: Overlapping Execution with CUDA Streams

Figure 20 illustrates the concurrency scheme used in optimizing the matching process. From this point on, such diagrams will be referred to as stream diagrams. The top row and bottom row represent two separate CUDA streams.

CUDA streams allow the issuance of multiple GPU events in parallel, such as data transfers and kernel launches. In general, it is good practice to make sure that streams are working on independent datasets so that no conflicts occur. In addition, if using page-locked memory, pages that can’t be removed from RAM, data transfers can be non-blocking; freeing the CPU while the transfer takes place.

The diagram separates activity into two streams, Stream A and B. Purple events denote CPU activity. One will notice that kernel execution and transfer of matches are non-blocking so that deltas and bitmap conversion can occur in parallel with the GPU. However, when HtoD transfers occur, there is no host CPU activity. In the case of HtoD transfers, blocking calls are use. In addition, a synchronization point is placed before the blocking calls. The blocking transfer and synchronization point are used for 1 main reason; the Tree Bitmap structure can be very large, as the number of nodes at each depth can grow exponentially. It may be inefficient or implausible to pin a large amount of memory for non-blocking transfers as it prohibits use of that memory for other purposes. To mitigate this, only a single Tree Bitmap is used (compared to 2 for chunks and match buffers) and it can be swapped in and out of memory as needed.
3.3.13 Hybrid Tree Bitmap

Unfortunately, the stream diagram featured in Figure 20 is a best-case scenario in which the kernel executes in the same amount of time as the CPU’s delta and conversion procedures. In reality, either the kernel or the CPU procedures will be a bottleneck to execution. For the CPU, some of the most expensive operations occur when dealing with the intermediate data structure; this includes conversion from trie to Tree Bitmap and adding new k-mers to the trie via `deltas()`. When optimizing, these places may be a good place to start.

Adding k-mers to a trie requires a new node to be allocated for every new symbol that is added to the trie. In addition, a memory read is needed for every traversal to a new node in the trie. Conversions require recursive breadth first access to group nodes into trie-nodes. The recursive nature of the conversion, in addition to a memory access for each node, causes sub optimal performance. A new intermediate structure should resemble the Tree Bitmap structure more closely in order to cut out inefficiencies in conversion, but still retain the ability to be easily edited.
Figure 21: Hybrid Tree Bitmap

The Hybrid Tree Bitmap (HTB) achieves the goals stated in the above paragraph. Each node in the HTB contains an inmap and exmap. In addition, each node can have up to 64 exiting transitions and represent up to 21 nodes. To this extent, a HTB node resembles a trie-node. However, each HBT node is connected to its parent node and children nodes via pointers.

Since each HTB node represents a group of nodes (inmap), the number of allocations is reduced by (at most) a factor of 21. Instead of allocating a new node, a bit is set in the inmap of the HTB node. Allocations of HTB nodes are made when an external traversal is made; in which case, the corresponding bit in the exmap is set.

The similarity between HTB and Tree Bitmap leads to lower overhead in conversions. Nodes no longer have to be grouped according to trie-node representation. For every breadth-first visit to a node in the HTB, the already existing inmap, exmap, and prefix_sum are pushed to the Tree Bitmap structure.
3.3.14 Levenshtein Automata and Adaption to Tree Bitmap

Chapter 1 discussed the implementation of Levenshtein Distance using a table and dynamic programming. That technique does not fit well in the use of automata. However, some of the ideas behind the dynamic programming implementation can be carried over and help develop an automata structure.

Figure 22: Hamming Distance Automata; For “act”

Roy et al. [23] discusses one NFA implementation for calculating Hamming Distance, a subset of Levenshtein Distance; it only calculates distance with substitutions. Here the diagonal transitions leading from one level to another represent substitutions by traversing on the complement of some character $c$ or $\overline{c}$. Each level in the automata represents a different Hamming Distance (i.e. each substitution increments the distance by 1).

The Hamming Distance automata is extended to the Levenshtein Distance automata in [24,25]. The core Hamming Distance automata remains the same while more states and transitions are added to represent insertions and deletions.
Figure 23 illustrates the Figure 22 automata with the addition of insertions. Similar to Figure 22, each increase in level represents an increase in the edit distance; the edit distance is represented by the superscript on the state enumerations. The next level can be reached via vertical ‘*’ transitions, a transition taken for all symbols. Thus, an insert causes a transition to the next level without losing place in the recognized sequence act.

Figure 23 also adds deletions to the automata, and thus, it is a true Levenshtein Distance automata. \( \varepsilon \)-transitions allow for up to \( D \) deletions in a row where \( D \) is the maximum Levenshtein Distance detected by the automata. The \( \varepsilon \)-transitions “skip” over transitions that would otherwise be required for a match. Each “skip” is representative of a deletion and as before, increases the edit distance by 1.

### 3.3.15 Removing \( \varepsilon \)-Transitions

In theory, the \( \varepsilon \)-transitions above work well. Unfortunately, they can be computationally expensive because they may require computation for the activation of multiple states in response to a single state being activated. Furthermore, they’re just not

![Figure 23: Levenshtein Automata; For “act”](image-url)
an intuitive computation. For these reasons, the \( \varepsilon \)-transitions are removed using a technique called \( \varepsilon \)-Closure [28].

![Diagram](image)

**Figure 24: Removing \( \varepsilon \)-Transitions Using \( \varepsilon \)-Closures**

The \( \varepsilon \)-closure of some state \( S \) is the set of states reachable by \( S \) via \( \varepsilon \)-transitions only. Figure 24 illustrates the automata shown in Figure 23 with states grouped into \( \varepsilon \)-closures. The \( \varepsilon \)-transitions are removed and now the accepting states must be redefined such that if one of the original accepting states belongs to a \( \varepsilon \)-closure, the new state defined by the \( \varepsilon \)-closure must also be an accepting state. The new condition for recognizing a match is shown by Equation 4.

\[
k \geq d + (K - D) \tag{4}
\]

Equation 4 defines the triangular region of accepting states such that if a state is active in said region, it is also accepting. It says the depth, \( k \), of the active state must be greater than or equal to the edit distance, \( d \), of the active state summed with the difference between k-mer length \( K \) less the maximum allowed edit distance \( D \).
Finally, the astute reader will notice that each closure-defined state should have exiting transitions for each state defined in the closure. Doing so would make the NFA much more irregular and difficult to handle given the regular nature of the Tree Bitmap. This addition is left for future iterations of the project.

3.3.16 Levenshtein Tree Bitmap

The Levenshtein automata can be generalized to also work with tries. The original Levenshtein automata is separated into logical edit distance levels (dist-levels), each level representing a different edit distance (as discussed previously). Additionally, each dist-level represents a duplicate sequence of the last. Instead of representing an dist-level using a sequence, it can also be represented using an entire trie.

![Figure 25: Levenshtein Trie](image)

Figure 25 provides a small-scale example of a Levenshtein Trie with an edit distance of 1. The tries at each dist-level are duplicates of one another, just as the sequences in the Levenshtein automata of Figure 23. Insertion transitions lead to a
node’s counterpart in the trie at the next dist-level. Substitutions are also similar to the Levenshtein automata in that they move diagonally to a new depth; given some symbol \( c \), the complement \( \overline{c} \) activates (diagonally) states in the next trie that are transitioned to via symbols in the set \( c \).

To clarify with an example, a given state \( S_i \) has a counterpart node \( S_{i+1} \), where \( i \) is the dist-level. \( S_i \) has transitions on symbols \( a, g, \) and \( t \) and thus \( S_{i+1} \) must also have transitions on symbols \( a, g, \) and \( t \) since it is the counterpart of \( S_i \). The states at the end of these transitions are represented by \( S_{ia}, S_{ig}, S_{it}, \) and \( S_{(i+1)a}, S_{(i+1)g}, S_{(i+1)t} \). If node \( S_i \) is active and receives symbol \( a \), node \( S_{ia} \) must be activated in turn. In addition, the nodes \( S_{(i+1)g} \) and \( S_{(i+1)t} \) must also be activated since they are the diagonals of \( S_i \) and their entering transitions belong to the set \( \overline{a} = \{ g, t \} \).

\( \varepsilon \)-transitions are omitted from this Levensthtein Trie structure and are replaced using Equation 4. This leads to the detection of only a subset of deletions, as stated in the previous section, but is necessary to keep the regular nature of the Levenshtein Trie automata.

![Levenshtein Tree Bitmap with Labeled Transition Types](image)

Figure 26: Levenshtein Tree Bitmap with Labeled Transition Types

Since the Tree Bitmap structure is only a representation of a trie, it follows that it can also represent the Levenshtein Trie as shown in Figure 25. A duplicate Tree Bitmap
structure lies at each dist-level. Just as transitions are represented implicitly by the
traditional Tree Bitmap, I-transitions and S-transitions (I for insertion, S for substitution)
are also represented implicitly by the Levenshtein Tree Bitmap (LTB).

Figure 26 depicts 3 types of transitions. Each is a transition from one level of the
LTB to the next. Internal transitions are omitted. The horizontal transitions from one
dist-level to the next represent both I- and S-transitions. I-transitions will activate a bit’s
counterpart bit in the counterpart trie-node. S-transitions will activate the diagonal-
complement bits. S-transitions also exist diagonally from one Tree Bitmap’s dist-level \( d \)
to the next Tree Bitmap’s dist-level \( d+1 \). These work similarly but must be taken using a
trie-node’s exmap since they are transitioning between levels. Finally, K-transitions are
the transitions described in the original Exact Match algorithm.

3.3.17 Levenshtein Kernels

The new LTB requires a new kernel to be executed that pays more resemblance to
the Naïve Exact Match kernel than the optimized Shuffle Match kernel. In fact, the LTB
kernels can be thought of as a generalization of the Naïve Exact Match kernel. The LTB
kernel requires \( D+1 \) (where \( D \) is the maximum edit distance) active bitmaps to keep track
of active states for each of the \( D+1 \) dist-levels. However, only a single Tree Bitmap
structure is needed (inmaps, exmaps, and prefix-sums) since the Tree Bitmap at each dist-
level is identical.

Substitutions make it implausible to implement LTB kernels in a similar manner
to the Shuffle kernels. Each active state is able to spawn a maximum of 3 more active
states due to substitution. This means there’s an exponential increase of active states as
the algorithm progresses and can be represented by $\sum_{i=0}^{k} 3^i$. It would be unruly to track each of these active states; instead the algorithm does an exhaustive search-and-transition over the entire LTB.

Between I & S traversals, I traversals are the simplest. They can be done using a simple bitwise-or operation.

```
trieNode.act_inmap[d+1] |= trieNode.act_inmap[d]
```

Figure 27: I-Transition Implementation

This transitions active states laterally to the next dist-level but maintains their relative position in the inmap.

S-transitions are similar to K-transitions in that they require transitions within the active inmap itself, and also external transitions to deeper levels.

```
generate_next_s_depth( inmap, tx ):
    word = 0
    foreach bit in inmap:
        word |= 0x1 << (bit<<4);
    return (word * 0xF) ^ (word << tx)
```

```
trieNode.act_inmap[d+1] |= s_transition_inmap( trieNode.act_inmap[d], tx )
```

Figure 28: Internal S-Transition Implementation

Figure 28 shows strong resemblance to `generate_next_depth()` in Figure 13 except for one main difference. A mask is generated for each group of children that has a ‘1’ in
it (word*0xF); XORing the mask with the result of the traditional transition gives us the complement of that transition, the internal S-transition. \textit{s\_transition\_inmap()} is identical to \textit{transition\_inmap()}, however, uses the new \textit{generate\_next\_s\_depth()} function. The result should then be applied to the trie-node’s counterpart trie-node in the \textit{d+1} bitmap.

```
1  exmap\_mask = exmap\_s\_transition( trieNode, tx )
2  foreach bit in exmap\_mask:
3      tn = equation2( trieNode, bit )
4      treeBitmap[tn].act\_inmap[d+1] |= 0x1
```

Figure 29: External S-Transition Implementation

Figure 29 utilizes these tactics in generating an exmap mask and applies it to the \textit{d+1} bitmap for external S-transitions. \textit{exmap\_s\_transitions()} operates in the same manner as \textit{exmap\_transitions()} however it uses the new \textit{generate\_next\_s\_depth()} function.

\textit{Naïve LTB Kernel}

From here on in, the kernels will be presented from a conceptual standpoint. With the knowledge of presented pseudo-code and explanation of previous kernels, the reader is assumed to know how the transitioning is implemented.
As stated before, the kernel does an exhaustive search over each trie-node. Trie-nodes are transitioned deepest-node-first and deepest dist-level first. Figure 30 depicts the order of making transitions. The algorithm starts at Figure 30a, executing all of the K-transitions; there are no I or S-transitions seeing that it is the last dist-level. Next, on dist-level 0 (not to be confused with Level 0) the I- and S-transitions are executed in Figure 30b. The I and S-transitions must be executed before K-transitions of the same dist-level due to I & S-transitions dependency on the current state. Finally, Figure 30c executes the remaining K-transitions.

At each stream symbol, the kernel records matches and mismatches if they exist. Mismatches are only recorded if the kernel is at dist-level 0. Matches are recorded as
long as active bits meet the criteria of Equation 4. An optimized match format is used to represent matches and mismatches.

<table>
<thead>
<tr>
<th>Type</th>
<th>Stream Index</th>
<th>Bitmap</th>
</tr>
</thead>
</table>

Figure 31: New Match Format

The 32 most-significant bits of the match record the same information as the traditional format. The new format removes the depth and trie-node members. The trie-node member is unnecessary and depth is only needed for mismatches.

The new bitmap applies to both matches and mismatches. For matches, it records all of the found edit distances for a given symbol in the stream; each bit represents an edit distance corresponding to its location in the bitmap and thus may represent a distance between 0 and 31. Similarly for mismatches, each bit represents a depth at which the mismatch occurred. Each thread will update the match or mismatch bitmap using an atomicOr() operation to avoid any race conditions on the match/mismatch.

Transition Efficient Kernel

The first step in optimizing the naïve kernel is analyzing the transitioning scheme. The naïve kernel visits some dist-level \(d+1\), transitions it using K-transitions, and then iterates to dist-level \(d\). However, dist-level \(d\) also updates dist-level \(d+1\) using I & S-transitions. This means that trie-nodes at dist-level \(d+1\) are accessed multiple times, once for \(d+1\)’s K-transitions and then for \(d\)’s I & S-transitions.
Figure 32: Transition-Efficient Kernel Transition Order

Figure 32 displays the order in which transitions are executed for the efficient kernel. First, in Figure 32a, the kernel starts at dist-level $d$ but also reads $d+1$. Together, all I, S & K-transitions are executed that would update dist-level $d+1$. Figure 32b then iterates to the next dist-level $d-1$ and executes all I, S & K-transitions to $d$. In the final iteration, K-transitions are executed for the last level. In the naïve kernel, $D+1$ iterations exist; however in the efficient kernel only $D$ iterations exist. Matches and mismatches are handled the same.
Transition Efficient Kernel with Queues

The efficient kernel optimizes by re-ordering the execution of transitions. However, inefficiency still remains in how external transitions (transitions between levels) are carried out. It should be noted that this implementation keeps the core structure of the efficient kernel and its transition pattern. The only difference is how external transitions are implemented. The implemented algorithm is inspired by work queues for use in tree-based algorithms [26,27]. Figure 33 illustrates external transitions.

![Figure 33: External Transition Parallelism](image)

The figure depicts a simple Tree Bitmap structure. Two threads operate on the structure in a similar fashion to a CUDA warp (in lock-step). \textit{Thread}_0 operates on $\text{trie-node}_1$ and \textit{thread}_1 operates on $\text{trie-node}_2$. To reiterate, $\text{trie-node}_i$ is contiguous with $\text{trie-node}_{i+1}$ in memory. The bold transitions represent transitions taken given some symbol $c$. The two threads iterate through their respective external transitions in lock-step.
Figure 34a depicts the memory access pattern as the two threads iterate over each external transition. Despite each external transition being contiguous with other external transitions, the access pattern is un-coalesced and inefficient. A queue is implemented to take advantage of the locality of trie-nodes represented in Figure 34b. Instead of writing the trie-nodes themselves to the queue, indices of the trie-nodes are used. The end result is more coalesced memory accesses.

```
for each thread in warp:
  bcastTrieNode = __shfl( trieNode, thread )
  bcastExmapMask = __shfl( exmapMask, thread )
  writeBallot = __ballot( (1 << laneID) & bcastExmapMask )
  queueIndex = __popc( ((1 << laneID)-1) & writeBallot )
  queue[queueIndex] = equation2( bcastTrieNode, laneID )
```

Figure 35: Writing to Queue Implementation
The queue is located in the shared memory of an SM. To prevent intra-block synchronization, each warp is designated its own queue; synchronization between warps is much simpler. A write to the queue is shown in Figure 35. The main goal behind queue writing is to broadcast each thread’s external transitions, and have each thread queue the transitions in parallel.

Line 2 & 3 broadcasts a given thread’s trie-node and exmap mask to all other threads in the warp using \_\_shfl\(). The threads then decide which threads will be writing to the queue using \_\_ballot\() on line 4. On line 6 the queue index is calculated by viewing how many preceding threads in the warp are also writing. Using Equation 2, the correct index of the trie-node is written to the queue. At some later point in time the queue will fill and all threads in the warp will empty it in a coalesced fashion.

### 3.4 Experimental Results

Experiments were carried out on a server with 24 cores and a Tesla K40c NVIDIA graphics card with Kepler architecture. The hardware environment is organized in the tables below.

<table>
<thead>
<tr>
<th>Graphics Card</th>
<th>Tesla K40c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
</tr>
<tr>
<td>Cores per SM</td>
<td>192</td>
</tr>
<tr>
<td>Global Memory</td>
<td>11 GB</td>
</tr>
<tr>
<td>Shared Memory per Block</td>
<td>48KB</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon E5</td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
</tr>
<tr>
<td>Cores</td>
<td>24</td>
</tr>
<tr>
<td>Clock</td>
<td>2.40 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GB</td>
</tr>
</tbody>
</table>

Table 1: Hardware Configuration

51
The Tesla K40c is Kepler architecture with compute capability 3.5, making it the minimum architecture required to carry out shuffle-type operations. It has 15 streaming multi-processors (SM) with 192 cores per SM, allowing 6 warps to execute in parallel per SM. CPU operations are executed on one of 24 cores running at 2.40 GHz and a total of 64 GB of memory.

The results use randomly generated datasets of variable length. Results that coincide use the same randomly generated data set.

### 3.4.1 Kernel Optimizations

![Chart showing performance comparison between exact match shuffle and naive kernels](image)

**Figure 36: Exact Naïve vs. Shuffle Results**

Figure 36 shows the performance gain of the exact match shuffle kernel against the naïve kernel. It illustrates how performance is affected by an increasing value of $k$. The shuffle kernel varies very little with an increased value of $k$. The naïve kernel,
however, is dependent on $k$ or more specifically, the number of trie-nodes in the Tree Bitmap. At the naïve kernel’s worst, the shuffle kernel has a performance gain of about 3 orders of magnitude.

The shuffle kernel heavily outperforms the naïve kernel by removing dependency on the size of the Tree Bitmap. Instead, it is only dependent on the size of the input stream. Unfortunately, the current design of the shuffle kernel is still less than optimal. This is depicted via the *Warp Efficiency* metric, which measures the average warp utilization of the kernel. As expected, as $k$ increases, the warp efficiency also increases due to more threads in the warp transitioning and shuffling trie-nodes to the next thread. Warp utilization remains low because of the inherent inability of threads in a warp to act independently from one another. Further optimizations to this kernel will be explored in future iterations of the project.

![Figure 37: Levenshtein Naïve vs. Transition Efficient Results](image)

Figure 37: Levenshtein Naïve vs. Transition Efficient Results
Figure 37 depicts the results of the transition-efficient kernel against the naïve kernel. The blue bar represents execution time (seconds) of the transition efficient kernel on different k values (6, 9, 12) and the red bar represent the naïve kernel. The kernel was executed on a stream size of 500,000 with a seed size of 75,000 base pairs. The secondary axis shows the percent improvement in overall memory accesses.

The transition-efficient kernel optimizes on the naïve kernel by reordering I-, K-, & S-transitions and executing transitions together that affect the same trie-node. By doing this, the efficient kernel is reducing the number of Global Memory reads and writes that the kernel is executing. The graph illustrates that as $k$ is increased, the performance improvement of the efficient kernel increases over the naïve kernel. However, the reader will notice that even though relative performance increases, the memory improvement decreases from ~20% to ~17%. Despite this, the relative improvement increases due to the fact that many more memory accesses are made at higher $k$’s.
Figure 38 compares performance of the queue to the transition-efficient kernel implementation. The vertical bars represent the execution time, in seconds, of the respective kernel for a stream size of 100,000 for varying $k$ values. It then relates the results to the change in occupancy from the efficient to the queue implementation.

Compared to the transition-efficient kernel, the queue kernel performs poorly. This decrease in performance is attributed to the decrease in occupancy caused by the introduction of the queue. From Figure 38, the queue occupancy sees decreases in the range from 40-50% from that of the transition-efficient occupancy. This arises from the use of the shared memory queue. Figure 38 uses a queue size of 128 elements. This means that each warp is allocated 512 bytes ($128 \times 4$). On top of this, the queue implementation also requires extra registers for the queue. Between these two factors, the occupancy of the queue kernel drops drastically. The last bit of overhead that the
queue kernel encounters is the use of shared memory. Every access to a trie-node via an external transition requires an extra read and write to shared memory.

The overhead accumulated from the three factors above greatly outweighs benefits that might be received from the queue. In fact, it is clear now that the queue may not offer any benefits. It requires that executed external transitions be next to each other in memory. While external transitions will lie contiguously in memory, there is no promise that these contiguous transitions will be taken and thus un-coalesced memory accesses will occur either way.

3.4.2 Intermediate Tree Structure Optimization

![Performance Gain HTB vs Tree Construction](a)
Figure 39: Intermediate Data Structure Optimization

Figure 39 illustrates the increase in performance of the HTB over the generic tree structure using results from building the data structure and conversion from the data structure to the Tree Bitmap structure. Each uses different k (6, 9, 12) values to show how the depth of the tree structure affects its overall performance. Each x-axis represents the size of the stream used in building the structure. The stream size is representative of the size of the data structure; more k-mers are added to the tree as the stream size increases.

Figure 39a shows the speed up of tree construction of the HTB over the generic tree. The performance here decreases with an increase in k. An increase in depth will increase the number of total allocations done for either tree. The allocation and initialization done for a single HTB node outweighs that of a generic tree node. In HTB, children are stored in an array pointed to by the parent node; this means if a child is allocated, the child’s brothers and sisters are also allocated and initialized.
Figure 39b shows the speedup of the conversion process for HTB over the tree structure. For the most part, the performance gain doesn’t vary much. This is to be expected; unlike in construction, conversion does not require allocation, it is more of a traversal process. The HTB requires less memory reads during traversal because of its use of bitmaps to represent nodes compared to the tree structure, which requires a new node to be read for every traversal. Finally, the reader should notice the variability of results in Figure 39b for $k=12$. The execution time for HTB conversion is typically a steady increase as the size of the tree grows. On the other hand, conversion of trees can be variable based on the trees overall structure. If nodes are grouped together densely, there will be a spike in execution time required to convert them into a trie-node structure.

3.4.3 Serial and Concurrent Dynamic Construction

Figure 40 depicts the results of the system running the shuffle kernel. CUDA function calls are separated into 3 logical rows, Host-to-Device transfers (top), Device-to-Host (middle), and Kernel calls (bottom); the actual time taken to complete these tasks is represented by the small slivers of blocks in each row. HtoD transfers mainly consist of transferring the Tree Bitmap structure to the GPU.DtoH transfers consist of transferring matches back to the CPU. CPU execution time is represented by the space between
blocks. The results are gathered using a stream-size of 1 million randomly generated base pairs, and chunk sizes of 100,000 base pairs with $k=15$.

There are three interesting things that are happening in this figure, and they are all correlated.

*The distance (time) between kernel executions increases as execution continues.* The time between kernel call increases from 250ms initially, to a final value of 587ms. After each execution, the CPU takes time to compute deltas and convert the intermediate data structure to a tree bitmap structure. Consequently, the intermediate data structure grows after each kernel execution. Because of this, conversion time from intermediate to Tree Bitmap will increase as well.

*HtoD sliver sizes (time) increases as execution continues.* As the tree size grows it is obviously going to require more memory. Consequently, an increase in memory requires an increased amount of time to transfer to the GPU.

*DtoH sliver sizes (time) decrease as execution continues.* With a small tree or trie, there are going to be a lot of patterns that exist in the stream but don’t exist in the trie. This absence of patterns is going result in many more mismatches. As the trie grows, fewer and fewer mismatches will occur. In reality, the rate at which the number mismatches decrease may be higher than expected due to repeating k-mers in the typical genome [30].
Finally, the reader should notice the results of the CPU version that collected the same matches from the input stream. While the proposed system took about 3.4s to complete the CPU was able to gather the same information in 1.55s. The overhead of computing deltas plus conversion time provides a large penalty in overall execution time. The CPU does not incur these penalties because it operates on a single trie structure.

![Figure 41: Concurrent Execution Streams](image)

Figure 41 explores the results obtained using concurrent execution streams. Stream A and Stream B are two separate CUDA streams of execution. Each sliver of is similar to that of Figure 40. Figure 41 provides a zoomed in view of one of these slivers. It provides a close-up view of the typical HtoD &DtoH transfers and kernel execution. HtoD transfers are synchronous (blocking), while DtoH transfers are asynchronous (non-blocking).

The reader should first notice the execution pattern for each stream is similar to that in Figure 41; this is for the same reasons mentioned above. In the case of concurrent streams, the distance between kernel calls in a single stream is roughly twice that
depicted in Figure 40. This is because each space represents delta and conversion times for kernel calls in each stream or twice the number or two delta calls and two conversion calls. Without initialization times taken into account, the system executes in ~2.9 seconds compared to the serial time of ~3.4 seconds. At first it might appear that savings might come from latency hiding from the use of separate CUDA streams but this is not the case. The majority of savings comes from conversion times. Because of the concurrent nature, each kernel call uses an intermediate tree that has not yet been updated by the other kernel’s previous calculated deltas. For this reason, the tree is smaller and thus requires less time for conversion.

3.4.4 Conclusion and Future Work

The preceding results were gathered to show the performance of the currently standing system. The results show that the system has its own strengths and weaknesses. It is strong when it comes to exact matching and it implements an efficient intermediate data structure. The approximate-distance kernels showed improvement by re-ordering I-, S-, & K-transitions but fell flat in the queue implementation in which queues created overhead that outweighed the performance gains that might come under special circumstances.

The stream diagrams presented in section 3.4.3 illustrate some bottlenecks that the system has. In order to become a performant dynamic system, the latency introduced by CPU operations (even with HTB) must be reduced. Future iterations should focus on parallelizing the CPU workloads where possible. In addition, pinned memory could also be applied to the tree bitmap structure using Butler’s pinned memory buffers [29] to
avoid overhead of reallocating pinned memory. While the performance of this system is not quite up to par yet, it provides a foothold for future iterations.

Finally, for the system to operate using approximate distance matching, the kernels need to be optimized a step further. Future iterations should explore the possibility of 1) reducing the complexity of the kernels 2) exploring new access patterns and possibly new structures and 3) using full fledged Levenshtein Distance. Until the execution times for these kernels are decreased, they will remain a huge bottleneck for the system.
Chapter 4: Conclusion

This thesis followed the process of building a dynamic automata-based system for use in matching k-mers in DNA sequences. It explored different data structures, algorithms, and optimizations to be made. The following contributions were made:

- The adaptation of the Tree Bitmap structure to a structure that can be used efficiently on the GPU for exact matching problems. The Tree Bitmap structure was also adapted for use in approximate distance matching.

- The implementation and optimization of an intermediate data structure for use in dynamic construction of trie-based automata and the efficient conversion to Tree Bitmap based automata.

- The implementation and optimization of Kernels used in exact and approximated distance matching.

Finally, this thesis is meant to provide a foothold for further development in this area. Future iterations may want to generalize what was done here for use with any automata structure and even for use on other architectures.
References


[26] Wu, Hancheng, and Michela Becchi Da Li. "Compiler-Assisted Workload Consolidation For Efficient Dynamic Parallelism on GPU." (Queue)


