Accelerated Systems for Pattern Matching

by

Arun Subramaniyan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Arun Subramaniyan
arunsub@umich.edu
ORCID iD: 0000-0001-6119-3182

To my parents and grandparents

ACKNOWLEDGMENTS

Firstly, I feel deeply honored to have been advised by Professor Reetuparna Das. I am constantly amazed by Professor Das' enthusiasm and passion for research. Even before starting graduate school, Professor Das gave me a head start to my PhD by involving me in her compute cache research project. Since then, I have greatly enjoyed and benefitted from each of my interactions with her. Over the last 5 years, she has helped me think systematically and spent countless hours providing structure and clarity to several of my hazy ideas. This thesis would never have been possible without her constant support, encouragement and guidance.

I would also like to thank Professor Satish Narayanasamy, Professor David Blaauw and Professor Westley Weimer for agreeing to serve on my dissertation committee and providing many useful suggestions to improve this thesis. Professor Narayanasamy was always available to discuss different ideas and their merits. Professor Blaauw helped me navigate several research challenges and is a constant source of new ideas. Professor Weimer has provided valuable feedback to improve both my oral and written presentation. He also encouraged me to open source our genomics benchmark suite and provided helpful suggestions to improve its reach.

My sincere thanks to Dr. Bill Bolosky who hosted me for two summer internships at Microsoft Research. I will greatly miss my interactions with him. He always encouraged me to strive for high-quality work with long-term impact and provided many helpful suggestions on job search. I would also like to thank Dr. Ramarathnam Venkatesan for making my stay at Redmond memorable and providing me with many valuable life lessons.

I am also deeply grateful to my undergraduate research mentors at Karlsruhe Institute of Technology: Muhammad Shafique, Semeen Rehman, Florian Kriebel and BITS-Pilani: Nitin Chaturvedi. They were the first to introduce me to research and motivated me to pursue a PhD.

Much of the work in this thesis would not have been possible without inputs from my amazing collaborators. Jack Wadden and Kush Goliya contributed a lot to this thesis. I greatly enjoyed brainstorming with them and appreciate their critical feedback on different ideas. Daichi Fujiki and Tianjun Zhang are joint first authors with me on our GenAx, genome accelerator work. I thank them for allowing me to present the results of our collaboration in my dissertation. Sanchit Misra, Md. Vasimuddin and Somnath Paul from Intel Labs provided many useful inputs on the Genomics-Bench project. Kevin Skadron, Elaheh Sadredini and Reza Rahimi from the University of Virginia provided valuable feedback on Cache Automaton and helped in the design of our in-cache DPDA accelerator, ASPEN. I also appreciate Ezhil Balasubramanian for his help in designing Cache Automaton and Jingcheng Wang for discussions on SRAM-based switches. Many thanks to Kevin Angstadt for patiently answering my questions on parsing pipelines, grammars and automata.

I also greatly appreciate the generous support from my sponsors. Parts of this thesis were supported by Precision Health at the University of Michigan (Precision Health Scholar Award), by the Kahn foundation, by Rackham's International Student Fellowship, by the NSF under the CAREER-1652294 award and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Many thanks to Colin Hercus from Novocraft Technologies for providing an academic evaluation license.

A special thanks to all the members of the MBits and Precision Health Research Groups: Daichi Fujiki, Vidushi Goyal, Charles Eckert, Xiaowei Wang, Nathan Ozog, Xiao Wu, Tianjun Zhang, Zhehong Wang, Yu Zeng, Timothy Dunn, Harisankar Sadasivan, Yufeng Gu, Yichen Gu and Alireza Khadem for all the fun times, attending numerous practice talks, discussions, dinners and more. I would also like to thank all the members of the CSE department and staff for patiently answering my questions, booking rooms for practice talks, helping with group lunch orders and purchasing.

At the University of Michigan, I feel incredibly fortunate to spend time with an incredible group of friends and seniors. Thank you Subarno, Aman, Ankush, Aditya, Mani, Jana, Vivek, Vidushi, Harini, Mitali, Sandipp, Suresh, Sanjay, Subramaniam, Salar, Hossein, Amir, Subhankar and Aporva for making my time at Ann Arbor enjoyable. I also thank my seniors – Shaizeen, Supreet, Ram for guiding me in the initial years of my PhD. Finally, I am deeply indebted to my parents – Subramaniyan and Bharathi, late grandfathers – Gopalakrishnan and Raman, grandmothers – Radha and Ananthalakshmi for their unconditional love and support throughout the thesis, especially during the difficult times. I dedicate this thesis to them. I also thank my sister – Subhasri, brother-in-law – Afsar and my extended family of uncles and aunts for their continuous encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Pattern matching forms the core of many applications and contributes to a significant fraction of their execution time. For instance, scanning the human reference genome for motifs and identifying variations between individuals requires processing of 100s of gigabytes of unstructured data and can take several days on a multi-core processor. Parsing activities in the frontend of browsers can account for up to 40% of web page loading time. Datacenter log processing involves the analysis of data generated at the rate of several terabytes every few minutes.

While general-purpose processors have been optimized for regular, data-parallel workloads, the class of pattern matching workloads identified above typically employ computational models and data-structures that are not well suited for general-purpose processing. In particular, these workloads perform irregular memory accesses and spend disproportionately more time and energy in moving data from storage to compute units when compared to the actual computation and are often bottlenecked by memory bandwidth. To address these inefficiencies, this dissertation proposes to repurpose existing memories for efficient in-memory pattern matching computation and presents new hardware-software co-design techniques to significantly improve the performance of these pattern matching workloads.

First, a new hardware design is proposed that allows embarrassingly sequential finite state automata, a common computation model used for pattern matching, to be executed in parallel in a DRAM-based in-memory accelerator. Next, this dissertation presents the Cache Automaton architecture, which repurposes CPU last-level caches for massively parallel automata processing. This dissertation also takes a deep dive into accelerating genomics analysis, an emerging application domain which heavily relies on pattern matching. In particular, we design custom hardware and present new hardware-friendly data-structures and algorithms to accelerate read alignment, a time-consuming string matching workload in genomics analysis, which matches each of the billions of short fragments of DNA emitted by the sequencer (called reads) against a reference genome. Finally, this dissertation presents a detailed characterization of the pattern matching landscape in genomics and highlights opportunities for the development of new domain-specific architectures customized for genomics analysis.

CHAPTER 1

Introduction

By 2025, the International Data Corporation (IDC) estimates that the total amount of data produced by individuals and corporations would be 165 ZB (10^{23} bytes) [11]. A major contributor to this data is unstructured data, which is being generated in large volumes in forms such as system logs, social media posts, emails, and news articles. Unstructured data is also expected to grow at an annual rate of 61%. With growing volumes of unstructured data, it becomes increasingly important to develop fast and efficient pattern matching techniques that can parse this data and extract insights.

Pattern matching on unstructured data poses unique challenges to general-purpose processing. While CPUs have been optimized for data-parallel processing of regular structured data, many applications dealing with unstructured data often employ computational models that are not amenable to efficient general-purpose processing. For instance, finite state automata (FSAs) are widely used as the computational model in many end-to-end applications that utilize pattern matching such as deep packet inspection [181], web-browser frontend [93] and motif-search [148]. FSA computation is inherently sequential and hard to parallelize. Modern multi-core processors cannot efficiently process FSA (in particular non-deterministic finite state automata or NFA), since they are limited by the number of state transitions they can perform per thread in a given cycle and suffer from branch mispredictions. This limits the number of patterns they can identify. Furthermore, their processing capability is also limited by the available memory bandwidth. GPGPUs (General-Purpose Graphics Processing Units) on the other hand have had limited success with automata processing because FSA processing is inherently dominated by irregular memory accesses.

Therefore, there is a need for efficient computing architectures to accelerate FSA computation.

Pattern matching using FSA is dominated by data movement and involves very little computation per state transition. To address this inefficiency, Micron introduced the Automata Processor (AP) which facilitates in-situ FSA computation in DRAM in a highly parallel and energy-efficient manner. AP has been successful in accelerating FSA computation because of three factors: massive bit-level parallelism inherent in memory technologies such as DRAM, eliminating data movement overheads (between memory and CPU), and reducing instruction processing overheads compared to a CPU. While AP significantly improves the state-of-the-art, it still processes the input stream sequentially. To break the sequential execution bottleneck and enable parallel processing of a single input stream, we propose the Parallel Automata Processor (PAP) architecture that demonstrates the feasibility of enumerative parallelization on the AP with low-cost hardware extensions.

DRAM is an attractive substrate for automata processing because of its density. However, we notice that a significant fraction of the die area in the AP is devoted to custom logic for efficient state transition. As a result, the AP can sacrifice the density of DRAM storage by up to 16.6$\times$ to support automata processing. This is because DRAM technology has not been shrinking at the same rate as processor logic and implementing additional logic to support state transition on lower technology DRAM nodes is inefficient. To overcome this limitation, we explore the feasibility of repurposing the last-level cache in general-purpose processors for automata processing. This has several advantages: (1) SRAM is faster and more energy-efficient than DRAM, (2) SRAM is integrated on-chip and can benefit from being implemented in cutting-edge technology nodes, similar to the rest of the performance-optimized logic.

Having seen the benefits of in-memory pattern matching, we looked at application domains that can benefit from such acceleration. One application domain that heavily relies on fast and efficient pattern matching is genome sequence analysis. A genome is essentially a long string of characters or bases from the DNA alphabet i.e., `A, G, C`, and `T`. A single strand of the human genome has $\sim$3 billion characters. Current sequencing technologies cannot read the entire genome at once, and typically split the DNA into billions of small substrings called *reads*. A major computational problem in sequence analysis is the problem of mapping each of the sequenced reads to their original position in the genome prior to splitting, referred to as *read alignment* in literature. This is typically performed in two steps: (1) an exact string matching step called seeding, which involves performing exact matches of short substrings in the read against a previously sequenced reference genome, and (2) an approximate string matching step, called seed-extension which matches the rest of the read at the candidate reference locations identified by seeding, while allowing for edits. Approximate string matching is needed to account for sequencing errors and true variations between individual genomes.

The most widely used software for read alignment, BWA-MEM [118] and the recently released faster version, BWA-MEM2 [128], perform seeding using a compressed index data structure called the FMD-index, that supports only single character queries on the index. Furthermore, each of the index accesses tends to touch a different part of the index data structure and exhibits little spatial or temporal locality. This leads to high memory-bandwidth requirements and poor performance on conventional CPUs. To overcome this limitation, we propose a memory-bandwidth aware data structure for seeding called Enumerated Radix Trees (ERT) that is designed from the ground up to support multi-character lookup. We further redesign the seeding algorithm in BWA-MEM2 to exploit reuse opportunities inherent in the seeding algorithm.

On the other hand, approximate string matching on CPUs is commonly performed using the Smith-Waterman algorithm [154]. This algorithm has time and space-complexity $\mathcal{O}(MN)$, where M is the length of the read string and N is the length of the reference string. Often, the read and reference strings are similar and only alignments with less than K edits are interesting. In those cases, the runtime and memory space can be improved to $\mathcal{O}(KN)$ [160]. We propose a non-deterministic finite automata called Silla for approximate string matching. The space requirements of Silla scale quadratically with edit distance and not string length and its time complexity is $\mathcal{O}(N)$. Furthermore, when compared to the Levenshtein automata commonly used for approximate string matching, Silla is string-independent and hardware-friendly since all state transitions are local. It is also composable, lending itself well to a hardware implementation that can scale to larger edit distances. To leverage these properties of Silla, we design a hardware accelerator called SillaX. SillaX also supports other features required for read alignment such as affine gap scoring, and traceback of alignment path.

1.1 Parallel Pattern Matching : Parallelizing Input Stream Processing on In-Memory Hardware Accelerators

While the Automata Processor can perform several thousand state-transitions in parallel, it still processes the input stream sequentially by looking at one input symbol every cycle. Parallelizing across a single input stream can significantly improve throughput. Parallelization of finite state automata is known to be a hard problem due to its inherent sequential nature and high computational complexity. To break this sequential dependency and allow for parallel processing, enumeration may be employed [136]. In enumeration, the input is divided into segments and computations are done on individual segments in parallel. This computation is carried out for every possible state of the FSA. Once all input segments have been processed, their results can be composed to identify the true paths (i.e., sequence of states visited in the sequential execution of the complete input).

While the enumerative approach is promising, there are several challenges to realize it on the AP. First is the difficulty of tracking enumeration paths in a non-von Neumann architecture like the AP which has no notion of threads or local variables. Enumeration paths need to be tracked to discard false paths and retain true paths when combining the results from different input segments. The second challenge is the sheer computational complexity of enumeration. Enumerations can be highly inefficient because in the worst case each state has to be enumerated. Real-world NFA can have tens of thousands of states. In general, enumeration of an FSA with n states, over k input segments can lead to an ideal speedup of k provided we have n $\times$ k independent computing resources. For typical NFA, these resources far exceed what is available on the AP.

To solve the above problems, we leverage some unique properties of real-world NFA as well as unique features of the AP. For instance, some enumeration paths can be pruned by partitioning the input stream at symbols that have a small set of state transitions defined for them in the NFA. To solve the enumeration path tracking problem, we utilize the notion of AP "flows" which allows different users to time-share the same NFA execution on different input streams. A "flow" is defined as the set of active states for each user. We notice that the AP flow can be repurposed to track each enumeration path. Similar to prior work [136], we observe enumeration paths converge at runtime and implement lightweight dynamic convergence checks in AP using the flow abstraction. *The proposed parallelization scheme demonstrates significant speedup (25.5$\times$ on average) using a 4-rank AP (with 32 DRAM dies) compared to sequential execution averaged across several benchmarks spanning network intrusion detection, malware detection, text processing, protein motif searching, DNA sequencing, and data analytics.*

1.2 Cache Automaton for Pattern Matching

Given the benefits of memory-centric automata processing as demonstrated by the DRAM-based Automata Processor, this dissertation seeks to answer the question: Are SRAM-based last-level caches (LLCs) a suitable substrate for automata processing?

Caches typically have lower capacity compared to DRAM and one may wonder if it can store large real-word state machines. Interestingly, we observe that DRAM-based AP sacrifices a huge fraction of die area (up to 16.6$\times$) to support state transition logic on lower technology DRAM. Thus, while DRAM's packing density is high, DRAM-based AP's packing density is comparable to LLCs (20-40 MB) which are located on-chip and can benefit from performance-optimized logic.

The memory technology benefits of moving to SRAM are apparent, but repurposing the 40-60% passive on-chip LLC area for massively parallel automata computation comes with several challenges. A naive approach that processes an input symbol every LLC access ($\sim$20-30 cycles @ 4GHz), would lead to an operating frequency comparable to DRAM-based AP ($\sim$200 MHz), negating the memory technology benefits. Increasing the operating frequency further can be made possible only by two insights. First, architecting an in-situ computation model that is cognizant of the internal geometry of LLC slices. We observe that the LLC access latency is dominated by wire-delays inside a cache slice, accessing upper-level cache control structures, and network-on-chip. Fortunately, in-situ architectures require only SRAM array accesses and do not incur the overheads of a traditional cache access. We use sense-amplifier cycling techniques to further accelerate SRAM array access. We also leverage the internal geometry of LLC slices to build a hierarchical state-transition interconnect. *All these optimizations lead to a speedup of 12×-15× over AP on average across a wide variety of automata benchmarks from the ANMLZoo [166] and Regex [43] benchmark suites using a 40 MB LLC.*

1.3 Accelerating Pattern Matching in Genomics

Exact and approximate string matching find extensive utility in genomic data analysis. During primary analysis, a sequencing instrument splits the DNA molecule into *billions* of short ($\sim$101 bp) strings called *reads*. Secondary analysis aligns each of the reads to a reference genome and determines genetic variants in the analyzed genome compared to the reference.

Short read alignment is one of the major compute bottlenecks in secondary analysis [30], contributing 30% to the overall runtime. In read alignment, these reads are aligned by matching them to a previously sequenced genome. This task is complicated by the fact that the new individual's genome may not exactly match that of the reference genome. In fact, the end goal is to determine the variants in the new genome. Naively aligning by matching a string to every possible position in the reference genome is computationally intractable. Popular read aligners such as BWA-MEM [118] and Bowtie [112] solve this using seeding. Seeding finds a set of candidate locations (*hits*) in the reference genome where a read can potentially align. Hits for a read are determined by finding exact matches for its sub-strings (*seeds*) in the reference. The seed-extension phase then uses approximate string matching to select the hit with the best score as the alignment position for the read.

Both the seeding and the seed-extension steps of read alignment are important candidates for acceleration. Seeding contributes $\sim$40% to the overall execution time of BWA-MEM2 [128] and seed-extension contributes 35%.

1.3.1 Enumerated Radix Trees : Enabling Memory Bandwidth-Aware Exact String Matching

We focus on seeding in BWA-MEM2, as it is the fastest available implementation of BWA-MEM, which is recommended as industry standard in the Broad Institute's best practices pipeline. The primary performance bottleneck in seeding is memory bandwidth. This is because both BWA-MEM and BWA-MEM2 use a compressed index structure (FMD-Index) that only allows iterative processing of each base in a read, leading to high bandwidth requirements.

BWA-MEM trades off high memory bandwidth for small memory space by using a highly compressed FMD-index (4.3 GB for human genome). In contrast, we propose a data structure for seeding that makes the opposite trade-off: it trades off increased memory space for reducing bandwidth, while still fitting within a modern server's main memory (64 GB). BWA-MEM2 also makes a similar tradeoff but our solution further improves bandwidth efficiency ($4.3\times$) by virtue of supporting multi-character lookup and exploiting reuse opportunities present in the seeding algorithm.

We refer to our bandwidth-efficient data structure as Enumerated Radix Trees (ERT). Like FMD-index, ERT enables variable length exact match search functionality. But, unlike FMD-index, it avoids iterative lookup for every base on a large structure. It achieves this by coalescing all substrings in a reference genome that start with the same k-mer (string of length k, where k is less than the minimum length for a seed) together, and representing them using a variant of a radix tree. As we discuss later, ERT allows *multiple consecutive bases to be matched with one lookup*, and exhibits better spatial locality than FMD-index. ERT also helps *reduce computation when substrings* within a read that need to be matched with the reference *overlap* using a prefix-encoded radix tree.

ERT's increase in bandwidth efficiency unlocks significant acceleration potential. To exploit this acceleration potential, we design a custom seeding accelerator. The seeding accelerator leverages a butterfly network to efficiently feed data to parallel specialized seeding processors. Each seeding processor leverages light-weight context switching to provide high compute density and hide the long latency of DRAM accesses. *The FPGA ERT seeding accelerator can achieve up to 2.8 million reads/s on AWS F1 FPGA resulting in a speedup of 2.6× over BWA-MEM2 seeding.*

We open source the ERT software implementation for the benefit of the research community. ERT-based seeding is also integrated into BWA-MEM2 (ert branch: https://github.com/bwa-mem2/bwa-mem2/tree/ert)

1.3.2 SillaX : Approximate String Matching Acceleration

Using the hits obtained from seeding, the seed-extension step performs pairwise approximate string matching between the read and reference substrings at each of the hit locations to determine the hit with the best score as the read's alignment position. Approximate string matches are scored using an *affine gap function* [134, 77], which is based on the Levenshtein edit distance, but weighs different edit types (insertion, substitution and deletion) differently. Apart from the alignment score, it is also required to output the trace of edits needed to align the read at the chosen reference position. This final step is referred to as *traceback*.

We propose a non-deterministic finite state automata for approximate string matching called Silla (String Independent Local Levenshtein Automata) and a corresponding hardware implementation based on Silla called SillaX. Silla has been designed from the ground up to support an efficient hardware implementation. Unlike the Levenshtein automata, Silla is string-independent and hardware-friendly since all state transitions are only to neighbouring states reducing communication complexity.

SillaX processing elements (PEs) are organized in the form of a systolic architecture. Each SillaX PE is customized to support affine gap scoring and traceback necessary to be used for seed-extension. SillaX also compresses traceback paths by keeping a count of matches in each PE. The proposed SillaX accelerator is also organized as composable sub-grids for flexibility. Compared to alternate Smith-Waterman implementations for string matching [84], the processing elements in the SillaX edit-distance machine are 30$\times$ smaller. Furthermore, SillaX supports in-place traceback. Prior approaches either require external traceback memory or increase the time complexity of traceback to $\mathcal{O}(NlogN)$. *SillaX provides 62.9$\times$ speedup over optimized banded Smith-Waterman running on a 56-thread CPU.*

1.4 GenomicsBench: Characterizing the Genomics Computing Landscape

Genomics is at the forefront of the precision medicine revolution. Genome sequencing can help in early cancer detection [161], developing targeted therapies to different tumor mutations [95], identifying the causes of complex genetic diseases [186], assessing risk factors, and developing new drugs. For example, 42% of the drugs approved by FDA in 2018 were based on precision medicine data obtained from genome sequencing [29]. With the advent of portable and cheap sequencers, it is now feasible to test and monitor the emergence of novel infectious diseases such as COVID-19 [3] among our population and take timely action to prevent their spread.

Over the last decade, advances in high-throughput sequencing and the availability of portable sequencers have enabled fast and cheap access to genetic data. Of particular note, is the advent of third generation sequencing, which enables reading out longer sections of sample DNA, but with higher per-character error rates. For example, a single modern sequencer can produce several terabytes of data per day at the low cost of $100 per human genome. As a result, sequencing data is now being produced at a rate that far outpaces Moore's law and poses significant computational challenges on commodity hardware. For a given sample, sequencers typically output fragments of the DNA in the sample. Depending on the sequencing technology, the fragments range from a length of 150-250 at high accuracy to lengths in few tens of thousands but at much lower accuracy. To meet this demand, software tools have been extensively redesigned and new algorithms and custom hardware have been developed to deal with the diversity in sequencing data. However, a standard set of benchmarks that captures the diverse behaviors of these recent algorithms is lacking. We believe that the availability of such a benchmark suite will be crucial in facilitating future architectural exploration, specific to this rapidly growing important domain. Towards that end, we present the GenomicsBench benchmark suite which contains 12 computationally intensive data-parallel kernels drawn from popular bioinformatics software tools. It covers the major steps in short and long-read genome sequence analysis pipelines such as basecalling, sequence mapping, de-novo assembly, variant calling and polishing. We observe that while these genomics kernels have abundant data-level parallelism, it is often hard to exploit on commodity processors because of input-dependent irregularities. We also perform a detailed microarchitectural characterization of these kernels and identify their bottlenecks. GenomicsBench includes parallel versions of the source code with CPU and GPU implementations as applicable along with representative input datasets of two sizes - small and large.

CHAPTER 2

Background and Related Work

In this chapter, we first provide background on how finite state automata can be used for pattern matching. We then describe prior compute-centric and memory-centric approaches to accelerate finite state automata computation. Later, we present some applications of pattern matching to genomics and describe prior hardware and software approaches to accelerate read alignment, a computationally intensive step in analyzing genomic data.

2.1 Finite State Automata for Pattern Matching

Non-Deterministic Finite State Automata (NFAs) form the core of many end-to-end applications that utilize pattern matching such as data analytics and data mining [169, 45], network security [181, 72, 110, 131], bioinformatics [148, 62, 172], tokenization of web pages [136], computational finance [18, 36] and software engineering [142, 58, 38]. In web browser frontends, finite state automata computations can contribute to about 40% of the loading time for many web pages [93]. The *oligo_scan* routine used in Weeder 2.0, an open-source tool for motif discovery in DNA sequences contributes 30-62% of the total runtime [170]. In the *Apriori* algorithm for frequent itemset mining, NFA processing accounts for 33-95% of the execution time, based on the frequency threshold [168]. Prior work [190] has shown that without accelerating finite state automata operations, it is infeasible for these applications to achieve sustained performance improvement, no matter how well other parts of these applications are parallelized (Amdahl's law).

A Non-deterministic Finite Automata (NFA) is formally described by a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a set of states, Σ is the input symbol alphabet, q_0 is the set of start states and F is the set of *reporting* or *accepting* states. The transition function $\delta\left(Q, \alpha\right)$ defines the set of states reached by Q on input symbol α. The non-determinism is due to the fact that an NFA can have multiple states active at the same time and have multiple transitions on the same input symbol.

NFA computation entails processing a stream of input symbols one at a time, determining which of the current active states match an incoming input symbol (*state match*) and looking up the transition function to determine the next set of active states (*state transition*). Conventional compute-centric architectures store the complete transition function as a lookup table in the cache/memory. Since a lookup is required for every active state on every input symbol, symbol processing is bottlenecked by the available memory bandwidth. This leads to performance degradation especially for large NFAs with many active states. With limited memory bandwidth, the number of state transitions that can be processed in parallel is also limited. Converting these NFAs to equivalent deterministic finite state automata (DFAs) also cannot help improve performance since it leads to an exponential growth in the number of states.

2.2 Automata Processing Acceleration

2.2.1 Software Approaches

Conventional compute-centric architectures like CPUs and GPUs typically store NFAs as a state-transition matrix in cache/memory. These architectures have two main limitations: (1) need for high memory bandwidth or memory capacity especially for large NFA with many active states and (2) high instruction processing overheads per state transition (as many as 24 *x86* instructions for a single DFA state transition [44]). As a result, several CPU/GPU-based automata processing engines have either limited themselves to DFAs [44, 182, 41, 184] or have proposed optimizations that aim at reducing memory footprint and minimizing the memory bandwidth [183]. Several works have also explored automata-friendly cache or memory layouts [86, 173, 103].

To deal with the high cache miss rates and branch misprediction rates associated with the random access patterns of finite state automata, SIMD operations have been explored [136, 121]. Several speculative and enumerative parallelization approaches have also been proposed to speedup FSM processing [136, 190, 189, 145, 92]. Ladner and Fischer [111] parallelize deterministic FSMs (DFA) using parallel prefix-sums. Hillis and Steele [87] present an improved parallel prefix algorithm that reduces the execution time from $O\left(log\left(m\right) \times n^3\right)$ to $O\left(log\left(m\right) \times n\right)$ when executing on *m* processors. More recently, Todd and others [136] leverage classic parallel prefix sums to do enumeration of FSMs on modern hardware. The key contribution of their work is three fold: improving enumeration efficiency by reducing the dependence on *n* (number of states in the FSM) by cleverly leveraging convergence, demonstrating a scalable implementation on modern multi-core processors with vector SIMD units and careful data mapping of the transition table based on the range of input symbols to improve the spatial locality of cache accesses. However, their work is limited to small DFAs, primarily due to the large computational complexity of enumerating NFAs.

To reduce the computational complexity and space footprint of conventional NFAs in multi-core architectures, modular NFA architectures have also been proposed [178]. To improve locality of access, several small regular expressions and regular expressions with *common prefixes* are merged into larger segments. Parallelism is achieved by mapping these segments to separate threads, with each thread processing either the same or different inputs in parallel.

An alternative to enumerating all states is *speculation*, i.e. guessing the start states of input segments [190, 189, 145]. Speculation for parallelizing FSMs has been applied to specific application domains such as browser's frontend [94], JPEG decoder that uses parallel Huffman decoding [104], intrusion detection using hot state prediction [123], and speculative parsing [96]. Notably, Zhao and others [190, 189] introduce the concept of principled speculation, which is the first rigorous approach to speculative parallelization.

2.2.2 Hardware Approaches

Compute-centric architectures: While several regular expression matching and DFA processing accelerators designs have been proposed in literature [162, 159], we discuss the two most relevant and recent designs. In general prior compute-centric hardware accelerators are limited by the number of parallel matches and transitions they can support. HARE [76] is a regular expression accelerator that is designed to match the DRAM memory bandwidth. However, the maximum number of simultaneous components that can be matched is limited to 64 even in the design with width = 32. This comes at an area cost of $80mm^2$. This limits its applicability for packet inspection which requires simultaneously matching against a large number of regular expressions ($>$3000). Also, its power grows quadratically with the number of patterns to be matched. On the other hand, the Unified Automata Processor [68] processes multiple input streams in parallel using vector lanes. However each lane is provided with only a 16 kB local bank, limiting the largest connected component in the NFA it can support. Furthermore, only 8 concurrent activations per lane are allowed and NFAs with many active states overflow the combining queue used in their design. This is a limiting factor for benchmarks that have hundreds to thousands of active states every cycle.

Memory-centric architectures: Micron's Automata Processor (AP) is an in-situ memory-based computational architecture targeted at accelerating automata processing and can facilitate highly parallel and energy efficient processing of finite state automata in hardware. Memory-centric architectures like the Micron Automata Processor (AP) [64, 167] are attractive for automata processing because the inherent bit-level parallelism of DRAM enables it to support multiple parallel state matches at bandwidths that far exceed the available off-chip pin bandwidth. Since the same input symbol can be matched against multiple states in parallel, instruction processing overheads are also reduced.

AP accelerates finite state automata processing by implementing NFA states and state transitions in memory. Each automata board fits in a DIMM slot and can be interfaced to a host CPU/FPGA using the DDR/PCIe interface. Figure 2.1 illustrates the automata processor architecture.

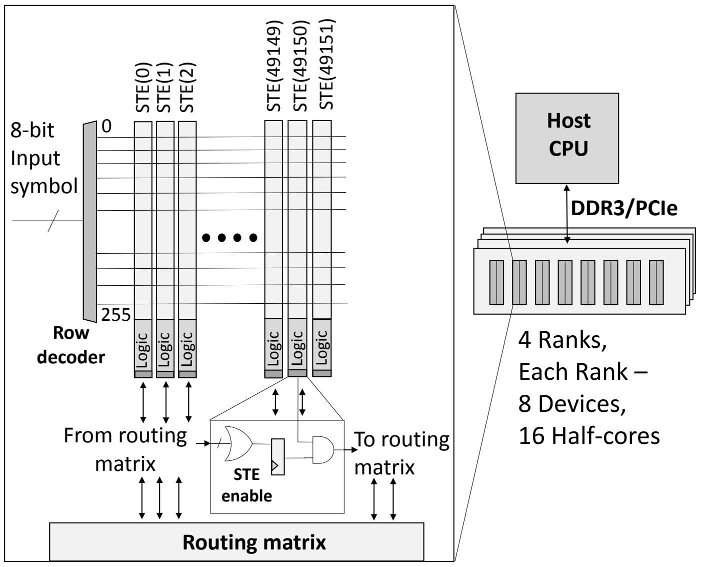


Figure 2.1: Automata Processor interfaced with the host CPU.
The architecture uses DRAM columns to encode automata states and custom logic to encode state transitions in a routing matrix. The address bus is repurposed to stream input symbols.

Before processing in the AP, the classical representation of NFAs needs to be transformed to a compact ANML NFA representation [64] where *each state has valid outgoing transitions for only one input symbol*. Thus, each state in an ANML NFA can be *labeled* by one unique input symbol. ANML NFA computation entails processing a stream of input symbols one at a time. Initially, all the start states are active states. Each step has two phases. In the *state match* phase, we identify which of the active states have the same label as the current input symbol. In the *state transition* phase, we look up the transition table to determine the destination states for these matched states. These destination states would become the active states for the next step.

In the AP, the FSM states (called State Transition Elements or STEs) are stored as columns in DRAM arrays (256 bits). Each STE is programmed to the one-hot encoding of the 8-bit input symbol (same as it is *label*) that it is required to match against. For example, for an STE to match the input symbol a, the bit position corresponding to the 97^{th} row must be set to 1. Each cycle, the input symbol (ASCII alphabet) is broadcast to all DRAM arrays and serves as the *row address*. If an STE has a '1' bit set in the row, it means that the label of the state it has stored matches the input symbol. *State match* is then simply a DRAM row read operation, with the input symbol as the *row address* and the contents of the row determining the STEs that match against the input symbol. Thus, by broadcasting the input symbol to all DRAM arrays, it is possible to determine all the states that match the current input symbol in parallel. *State transitions* between currently active states to next states are accomplished by a proprietary interconnect (*routing matrix*) which encodes the transition function. Reconfiguring this interconnect requires a costly recompilation step. Only the states that match the current input symbol and are active, undergo state transition. The bits of a register (*active state mask*) at the bottom of the STE columns determine the set of STEs that are *active* in a particular symbol cycle. These bits are initially set for only *start states*. All active bits for all STEs can be independently set in a given cycle, as they are all mapped to different columns of the DRAM arrays. Thus, the AP allows any number of transitions to be triggered in a given cycle, enabling massive parallelism and efficient NFA processing.

Due to physical routing constraints, each logical AP device (D480) is organized hierarchically as half-cores, blocks, rows and STEs with no state transitions across half-cores. Therefore, each half-core can be considered as the smallest unit of parallellization for partitioning into input segments. STEs configured as *reporting* have no outgoing transitions and their results are communicated to the CPU by writing to an *output event buffer*. Each entry in this buffer contains a report code and a byte offset (in the input stream) of the symbol causing the report. These entries are parsed in the host and communicated to the user. The current generation AP contains 4 ranks of 8 D480 devices each. Each device consists of 2 half-cores encompassing 49152 STEs, organized into 192 blocks. Each block further contains 256 rows and each row stores 16 STEs. The Micron AP also includes block-level power gating circuitry that disables a block with no active states. In terms of the reporting hierarchy, each AP device is also partitioned into 6 output regions, with each output region storing a maximum of 1024 reporting elements. Also present are 768 counters and 2304 programmable boolean elements to augment pattern matching functionality.

Recent efforts at the University of Virginia's Center for Automata Processing have demonstrated that AP can outperform x86 CPUs by $256\times$, GPGPUs by $32\times$, and accelerators such as

XeonPhi by 62$\times$, across a wide variety of automata benchmarks [166].

2.3 Pattern Matching in Genomics

Genomics analysis is one application domain that extensively employs pattern matching. Genomics refers to the study of the structure, function and development of genes and gene expression in an individual. There are several computationally intensive problems in genomics that involve extensive string matching. Some examples include identifying a sequence of base pairs and matching those to a database of known strings (e.g., motif search) or aligning a fragment of a query genome against a reference genome to identify potential similarity (e.g., genotyping). Whole genome sequencing (WGS) which determines the complete DNA sequence of an organism's genome is of particular interest in the near term because of its relevance to precision medicine [83], wherein strategies for disease prevention and drug selection are developed and customized to meet the needs of an individual. Since human genomes across individuals are more than 99.9% similar, sequencing the genome of a new person can be made faster by mapping it to an already sequenced genome (also known as the reference genome).

2.3.1 Common Genomics Pipelines

In this subsection, we provide a brief summary of some of the common genomics pipelines used to analyze reads from different sequencing technologies (both short and long read sequencing data) (illustration in Figure 2.2). Section 6 provides a detailed description of the different computational steps involved in these pipelines. All three pipelines start with the raw sequencer output. Given a biological sample, typically, multiple copies of the contained genome sequence are extracted and then decomposed into smaller nucleotide fragments. A sequencer reads the sequence of nucleotides in the fragments and generates raw signals based on what it reads. The first step in all the three pipelines prior to downstream analyses is the interpretation of these signals to derive reads, which are sequences of bases over the nucleotide alphabet {A,C,G,T}. This step is called **basecalling**.

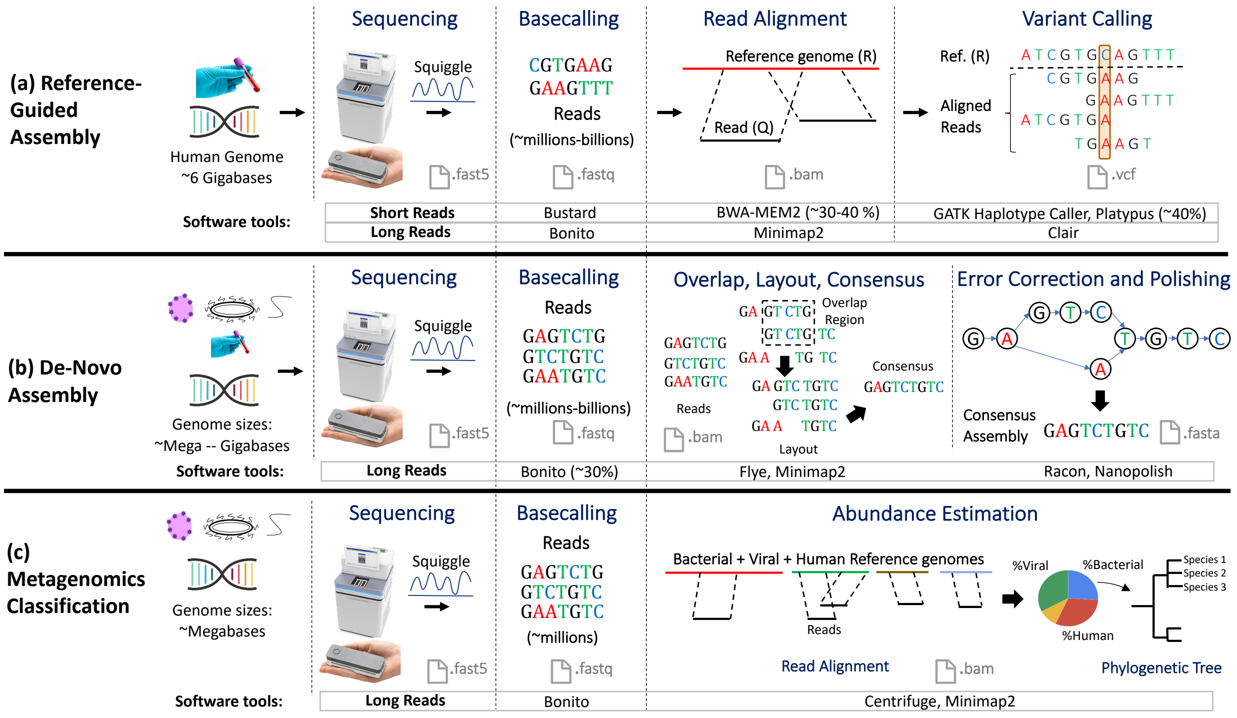


Figure 2.2: Common workflows in genomics.
(a) Reference-Guided Assembly. (b) De-Novo Assembly. (c) Metagenomics Classification

For Illumina sequencing machines, the signal data are fluorescence images which are converted into bases using a proprietary basecaller, Bustard [47]. For Oxford Nanopore (ONT) sequencers, raw signals are the current perturbations in the nanopore (e.g., in the FAST5 format). Guppy [175] is ONT's proprietary basecaller software. We characterize the open-source research basecaller from ONT, Bonito [4] as part of the nn-base kernel in GenomicsBench (Section 6.2), which demonstrates higher basecalling accuracy than Guppy [22].

Reference Guided Assembly: This pipeline reconstructs the sample genome by aligning reads from it to a reference genome and identifies differences in the sample (also called *variants*) compared to the reference genome. Typically, small differences, i.e., substitutions, short insertions and deletions (< 50 bases) are identified. Sufficient number of copies of the sample genome need to be sequenced to ensure random sequencing errors can be distinguished from true variations (each genome position is covered $30 - 50\times$ on average). This is especially needed for long reads from PacBio and ONT which have $5 - 15\%$ error rate per base [175, 174], resulting in input datasets of several hundreds of gigabytes. Subsequent analysis of this data can take several days on a modern multi-core processor [153].

Figure 2.2 (a) shows the two main time-consuming steps: **(1) Read Alignment**, which determines the best location for each read in the reference genome. **(2) Variant Calling**, which uses machine learning or statistics-based models to gather support for variants from aligned reads. BWA-MEM [118, 128] and GATK Haplotype Caller [8] are the most popular short-read software tools for these two steps recommended as part of GATK Best Practices [9]. These account for $\sim$30-40% and $\sim$40% time of the reference guided assembly pipeline respectively [128, 165].

De Novo Assembly: This pipeline attempts to assemble the reads into a genome *de novo* based on read overlaps in the absence of a suitable reference. The availability of long reads for de novo assembly has greatly improved the quality of draft reference genomes. This is mainly because they can span large structural variations (e.g, > 50 bases insertions/deletions, large rearrangements between the sample and reference genomes) [126] and can help resolve mutations from maternal and paternal chromosomes [151]. Long read de novo assembly is typically done using the *overlap-layout-consensus* method as shown in Figure 2.2 (b). In the overlap identification step, common seeds shared between read pairs are used to identify potential overlapping regions. In the layout step, these overlapping regions are extended into larger contiguous regions. Finally the consensus step corrects small errors in assembly. Large assembly errors are corrected in a later graph-based polishing step. For long-read assembly and polishing, Flye [105] and Racon [163] are popular software tools. Assembly of the human genome using Flye [105] and Racon [163] takes $\sim$4.5 days on a 64-thread server, each contributing $\sim$30% to the overall time [153]. Basecalling is performed using Guppy [175], ONT's proprietary basecaller, and also accounts for $\sim$30% of overall time [153].

Metagenomics Classification: The advent of portable sequencers like ONT MinION [27] has enabled several applications like real-time pathogen detection [146] and microbial abundance estimation [115] in the field. Abundance estimation involves aligning input microbial reads to a reference pan-genome (consisting of reference genomes of all bacteria, virus, fungi and humans) and later estimating the proportion of different microbes in the sample as shown in Figure 2.2 (c). It is typically performed using software tools like Minimap2 [119] and Centrifuge [100].

2.3.2 Read Alignment

In this dissertation, we focus on accelerating the read alignment step, which is one of the computationally intensive parts of the reference guided assembly pipeline ($\sim$30% of overall analysis time in GATK's best practices workflow [9]). Read alignment essentially maps a large number of sequenced reads to a reference genome using a combination of exact and approximate string matching algorithms. Read alignment is hard and time consuming because of the need to account for true variations between individuals (i.e., substitution, insertion, deletions of bases) [156], to detect and tolerate sequencing errors, the presence of a large number of repeated sequences ($\sim$50% of human genome [152]) and structural variants usually associated with abnormalities.

Since scanning the reference genome to identify the location for each read is computationally infeasible, state-of-the-art software tools for read alignment [118, 112, 125] use the seed-and-extend heuristic. Seeding identifies a set of candidate locations (hits) in the reference genome where the read can potentially align, by querying an index of the reference genome for the locations of short exact matches (seeds) from the read. The hits are then verified by an approximate string matching step called seed-extension, that extends these 'seeds' in both directions while allowing for gaps (i.e., insertions or deletions) and mismatches.

2.3.2.A Seeding

FMD-index [116], suffix-trees, suffix arrays [63] and hash-tables [130] are the most commonly used data structures for seeding. We focus on FMD-index based seeding in this dissertation since it is used in BWA-MEM/BWA-MEM2, which is recommended as part of GATK's Best Practices pipeline. While this work focuses on FMD-index based seeding, there exists a rich body of work that also uses hash-tables for seeding [35, 98, 177, 185] and have optimized its cache behavior [81, 82].

FMD-Index: To identify seeds and their locations in the reference genome, popular read alignment tools like BWA-MEM/BWA-MEM2 use a highly compressed data structure called the FMD-index [117, 71] which is built using both strands of DNA. As shown in Figure 2.3 (a), the FMD-index consists of: (1) the *suffix array (SA)*, which contains the locations of lexicographically sorted suffixes of the reference genome R, (2) the *Burrows Wheeler Transform (BWT)*, computed as the last column of the cyclically sorted suffix array of the reference, (3) the *count table (C)* which stores the number of characters in R lexicographically smaller than a given character c and (4) the *occurrence table (Occ)* which stores the number of occurrences of a character up to a certain index in the suffix array. Using the count and occurrence tables, one can identify intervals (s and e) in

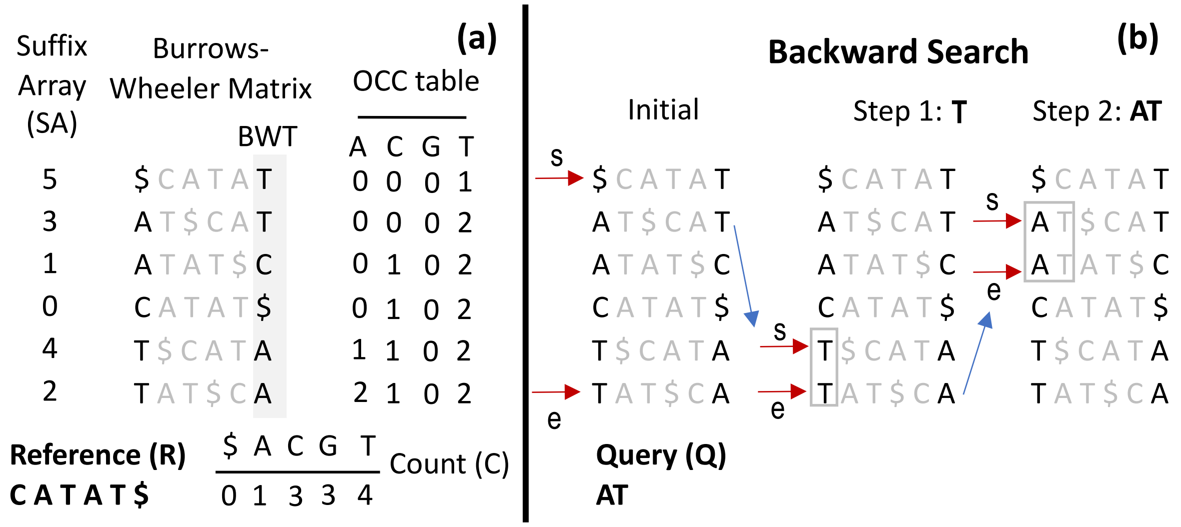


Figure 2.3: FM-index example.
(a) FM-index for reference R (b) Backward search using the FM-index.

the Burrows-Wheeler matrix where a particular query string exists in the reference by performing iterative lookups for each successive character in the query as shown in Figure 2.3 (b).

CPU-/GPU-based Seeding: FMD-index based seeding [117, 71] involves many irregular memory accesses and has been found to be bottlenecked by LLC and TLB misses on CPUs [49, 187]. Prior work has explored reordering memory accesses [187] and performing n-character lookup on an n-step FMD-index [49] to improve the locality and data requirements of FMD-index based seeding. Data-parallel architectures such as GPUs have also been leveraged to accelerate FMD-index search [48] by virtue of their high memory bandwidth and memory level parallelism.

Seeding Accelerators: Seeding accelerators based on the FMD-index use custom bit-wise operations to traverse the index and improve memory parallelism [171, 52, 60]. However, these implementations soon hit the memory-bandwidth roofline. Several other read alignment accelerators have also been proposed in literature [1, 127] that use hash-tables (with fixed-length keys) in novel ways to perform variable-length seeding. However, hash-based seeding algorithms produce a large number of seeds and hits that need to be verified by seed-extension and often need to be coupled with filtration algorithms [101, 176] to achieve comparable accuracy to variable-length seeds produced by FMD-index seeding.

2.3.2.B Seed-Extension

The fundamental operation in seed-extension is approximate string matching [155, 185]. Dynamic-programming and automata-based approaches are commonly used for approximate string matching.

Dynamic Programming: The most widely used algorithm in sequencing software is a dynamic programming algorithm called Smith-Waterman [154]. It computes optimal local alignments between two sequences by comparing segments of all possible lengths. It operates in two phases. *Score-computation* builds the dynamic programming matrix (N^2) based on a general scoring scheme. Then *traceback* constructs the optimal alignment by tracing back pointers starting from the highest scoring cell. It fundamentally has $\mathcal{O}(N^2)$ time and space complexity. While there have been several optimizations [70, 135], and approximation heuristics [188] developed to reduce their time, it does not scale well as string length increases.

Several FPGA-based hardware accelerators have been proposed for the Smith-Waterman algorithm [56, 127]. These leverage wavefront parallelism in systolic arrays to accelerate the *score-computation* phase of the Smith-Waterman algorithm. There has also been work on banded implementations of the Smith-Waterman algorithm [84], where only cells within a $2K + 1$ band around the principal diagonal of the Smith-Waterman matrix are computed. Most of these accelerators

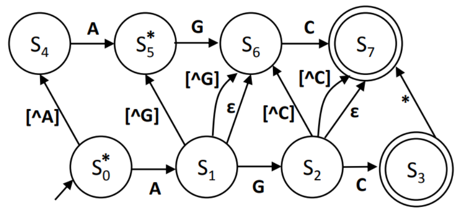


Figure 2.4: Levenshtein Automata for edit distance `K = 1` and reference string `AGC`.

also either offload the *traceback* phase to software or have traceback support only for short string lengths for an additional $\mathcal{O}(N)$ space overhead [139, 140, 127, 54].

Automata-based: The Levenshtein Automata (LA) for approximate string matching accepts all strings that lie within `K` edit distance of its stored pattern. Figure 2.4 shows an example LA. Each state essentially represents the position in the reference string up to which a match has been found, and the number of edits seen so far. As a result, it has a total of $K * N$ states. Its time complexity is $\mathcal{O}(N^2)$, as in the worst case all of its states may be active. Sequencing software systems rarely use LA based implementations as they struggle to outperform Smith-Waterman.

In-memory [65, 157] and ASIC automata accelerators [76, 69] can be used to implement LA. However, LA is poorly suited for hardware acceleration due to several reasons. One, since it is string dependent, the hardware needs to be reprogrammed every time the string changes, which can be prohibitive especially for seed-extension in sequencing. It requires processing billions of different reads, where each read needs to be compared to several seeds in the reference. Two, its space requirement is proportional to string length. When read lengths increase to millions of bases, LA based hardware solutions would be impractical. Third, none of the existing hardware automata accelerators support unique features required in sequence aligners: scoring, clipping, and traceback. It is challenging to include these features. For example, adding logic to compute affine gap scores for state transitions in Micron's Automata Processor (AP) is likely to be expensive.

A recent advancement in automata theory called Universal Levenshtein Automata (ULA) addressed some of the limitations of LA [133]. While ULA is string independent, it does not efficiently map to a hardware accelerator as communication between states are not local. Also, each state has a high-degree of fan-out ($\mathcal{O}(K)$), as every state in ULA is connected to a state in every higher level of edit distance to support deletions. To date, there is no hardware realization of the Universal Levenshtein Automata (ULA), nor has it been used in sequencing software.

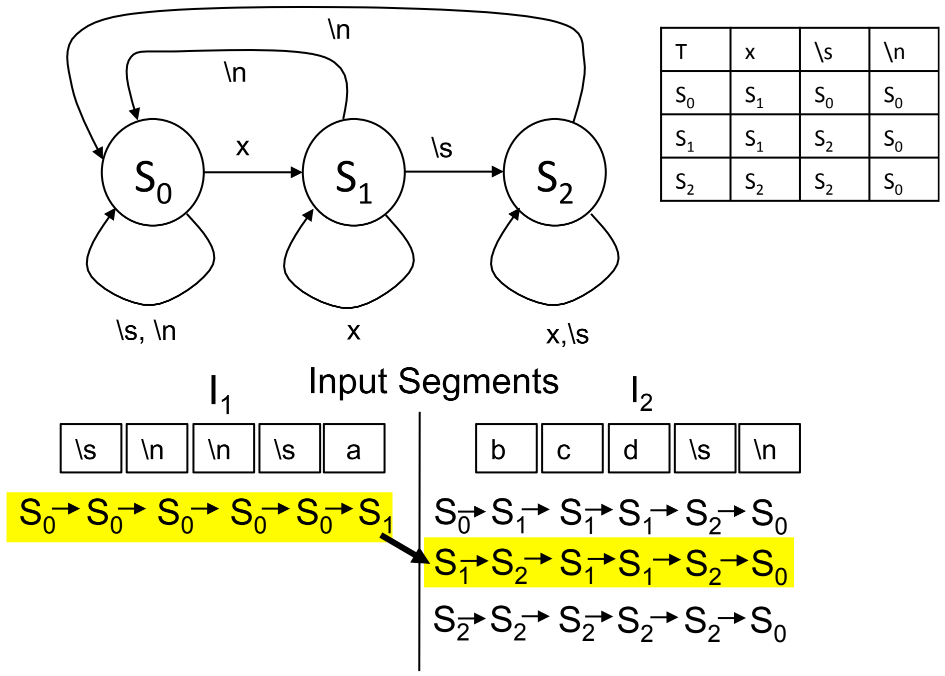
CHAPTER 3

Parallel Pattern Matching on In-Memory Hardware Accelerators

Spatial in-memory hardware accelerators like Micron's Automata Processor can accelerate automata processing by performing massively bit-parallel state-match and state-transition. However, they can still be bottlenecked by sequential processing of the input stream. To break this sequential execution bottleneck, enumerative parallelization is an attractive option. However, realizing enumerative parallelization in a non-von Neumann hardware accelerator substrate like the Micron Automata Processor comes with several challenges. In this chapter, we present these challenges and discuss architectural extensions to enable low-cost enumerative parallelization on the Automata Processor.

3.1 Finite State Machine Parallelization

Parallelizing finite state machine (FSM) traversal is known to be extremely difficult due to the inherent sequential nature of computation arising because of dependencies between consecutive state transitions. One way to parallelize FSM traversal is by partitioning the input string into segments, and processing these segments concurrently. This is feasible because FSM computation can be expressed as a *composition of transition functions* [87]. Parallelization is possible because transition function composition is *associative*. Figure 3.1 shows an example of FSM parallelization with two input segments (I_1 and I_2) each with five symbols. The FSM shown detects the first word in every line. The transition table is shown on right. Both these segments can be executed in parallel to provide a speedup of $2\times$ over the sequential baseline.


\n
\n
x
\s
S_0
S_1
S_2
\s, \n
x
x,\s
Input Segments
I_1
I_2
\s \n \n \s a
b c d \s \n
$S_0 \to S_0 \to S_0 \to S_0 \to S_0 \to S_1$
$S_0 \to S_1 \to S_1 \to S_1 \to S_2 \to S_0$
$S_1 \to S_2 \to S_1 \to S_1 \to S_2 \to S_0$
$S_2 \to S_2 \to S_2 \to S_2 \to S_2 \to S_0$

T	x	\s	\n
S_0	S_1	S_0	S_0
S_1	S_1	S_2	S_0
S_2	S_2	S_2	S_0

Figure 3.1: An FSM example with enumeration.
The FSM detects the first word in every line.

However, the starting states for each input segment, except the first segment are unknown. The starting states for the first input segment are the initial start states. The starting states for other segments are essentially the ending states of the corresponding previous segment. These dependencies prevent concurrent execution among threads. This problem can be solved by leveraging the classic parallel prefix-sum [111] algorithm. The basic idea is to execute the second segment for *every state* of the FSM. This method is referred to as an enumerative computation, since it involves processing each input symbol for all possible start states [136].

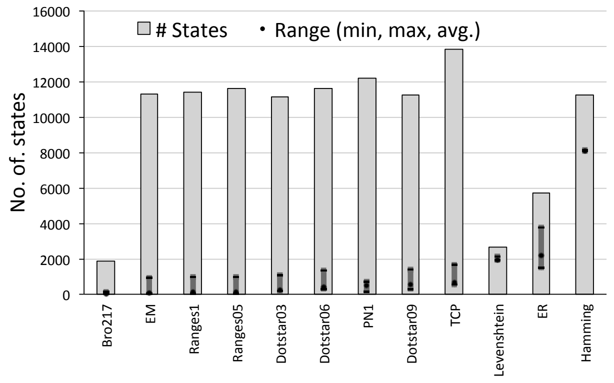
In Figure 3.1, the start state of the first segment is known (S_0) which is the start state of the FSM. However, the start states of input segment I_2 are unknown. Figure 3.1 shows an example enumeration for the second input segment, I_2. This example FSM has 3 states, so each segment (except the first) enumerates all 3 states. Once the first segment has finished, it can pick the correct or *true paths* from the enumerated paths of the second segment and discard *false paths*. Thus, the final result from the last input segment can be obtained by combining the intermediate results from all previous input segments. The true path for I_2 in Figure 3.1 starts at S_1, the remaining two paths are false paths. The final path of the FSM is highlighted.

The evident disadvantage of this method is the exponential blowup in computational complexity for processing each input segment. Consider a benchmark `Protomata`, an NFA which encapsulates 2340 known string patterns (called motifs) in protein sequences. Matching with known protein motifs can accelerate the discovery of unknown motifs in biological sequences in the field of bioinformatics. `Protomata` has 38,251 states. Enumerating all these states will make the parallel version orders of magnitude slower than the serial version.

Thus, unless we have the massive computational resources equivalent to n (i.e., the number of states in the NFA) $\times$ k (i.e., the number of input segments) processing units, enumeration can lead to slowdown instead of speedup. In this chapter, we explore different techniques for enumerating NFAs on AP's unique architecture and taming the computational complexity of enumeration. For instance, we find that the set of reachable states of an input symbol and the number of connected components can drastically reduce the number of enumeration paths. Similar to prior works on parallelization of deterministic finite automata (DFAs) [136], we find that many enumeration paths converge and design an AP architecture which is capable of near-zero cost dynamic convergence checks. The next section discusses the above and other optimizations which make parallelization of NFAs on AP profitable.

3.2 Parallel Automata Processor

This section discusses our proposed framework and architectural enhancements needed for effective parallelization of NFAs on the Automata Processor (AP).



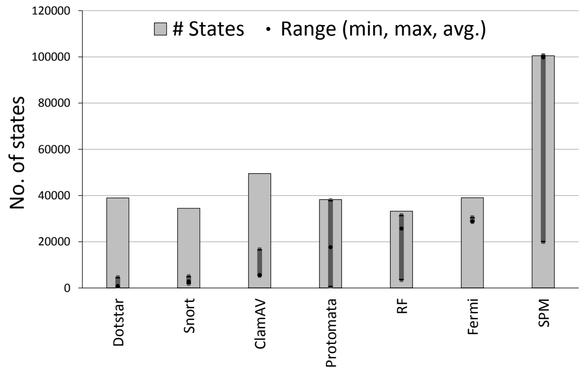


Figure 3.2: Range of symbols for different benchmarks.
Left: Smaller NFAs with state space limited to 15K states. ***Right:*** Large NFAs with greater than 15K states.

3.2.1 Range Guided Input Partitioning

Enumerating all states of an FSM will lead to exponential computational complexity. Fortunately, many of these states are impossible start states for a particular input segment. The *range of an input symbol* is defined as the union of the set of all reachable states, considering transitions from *all states* in the FSM that have a transition defined for that symbol. During actual execution, the range of the *last input symbol* in a particular segment determines accurately the subset of start states for the next segment. Any states outside this range are impossible start states. Our proposed parallelization framework partitions the input, such that, input segments end at frequently occurring symbols with small ranges to take advantage of minimum range input symbols. The symbol chosen for an FSM is obtained by offline profiling. Frequently occurring symbols are chosen to ensure that the size of input segments are roughly equal.

Figure 3.2 shows the average and minimum range across for the input symbols in the ASCII alphabet. Note that the AP only accepts 8 bit symbols, limiting us to 256 symbols. The bar depicts the total number of states in the system and the dark line indicates the minimum, average and maximum range for the 256 symbols. The figure demonstrates that the ranges of input symbols is a small fraction of total states, greatly reducing the complexity of enumeration. For instance, for `Protomata`, we can reduce the enumerated paths from 38251 start states to 667 start states. For some benchmarks the average range of symbols is almost as large as half the state space, example `SPM`. We discuss other optimizations for these benchmarks in the next section. Table 3.1 lists the range of the symbol chosen for input partitioning for each benchmark.

3.2.2 Enumeration using Flows

In this section, we provide an understanding of why we need flows to support enumeration, followed by a brief understanding of flows in the AP and how we can map enumeration paths to flows.

Ideally, one can activate all the start states and execute all enumerations simultaneously on one copy of the FSM. This is possible and *correct*, given that the AP seamlessly implements any number of simultaneous transitions in a given cycle and there is no limit on the number of start states that can be activated. This seems like a perfect solution, except that we lose all information about enumeration paths. After processing the input segments, we know what are the end states for *all* enumeration paths, but there is no way of knowing which path lead to which end states. Recall that after an input segment finishes execution, it will inform the next input segment which enumeration paths were the *true paths* and which paths were the *false paths*. The next input segment then must only use the results and end states of the *true paths* and discard the false paths.

In a conventional processor, enumeration paths are executed on SIMD threads and thread's local variables keep track of the start state of each path. In the AP, however there is no notion of local variables or state tracking. The only way to implement state tracking is by propagating the start state via the routing matrix. Routing matrix currently just routes 1 bit per state element pair (which encodes the transition between two state elements) and is already known to be a bottleneck in the system, both in determining the cycle time, and area complexity (occupies $\sim$30% of the chip). Augmenting the routing matrix with state information leads to exponential space complexity. Another possibility is replicating the FSM and executing the different enumeration paths in a separate replicated copy. Since each copy is mapped to a different half-core (or half-cores for large FSMs), we need as many half-cores as the number of enumeration paths. A typical AP D480 board has 64 half-cores which is far smaller than the number of enumeration paths for most of our benchmarks. This means that we can run at best one input segment at a time, leading to no speedup. Recall that speedup is proportional to number of input segments executing in parallel. Ideally, we would like to run 64 input segments one on each half-core and obtain a speedup of 64$\times$. Furthermore, mapping enumerations to different half-cores also complicates checking for convergence between the paths, because there is no path of direct communication between half-cores on different dies or ranks.

Our solution leverages AP flows to solve the above problem of tracking the start states of enumeration flows. Another unique advantage of using flows is that we can do low cost convergence checks, as we explain in Section 3.2.3.C. AP flows allow multiple users to time multiplex the AP for independent input streams. Each chip is equipped with a state vector cache which can store up to 512 *state vectors*. A state vector represents the state of an FSM execution and consists of 59,936 bits [(256 state enable bits per block + 56 counter bits per block) x 192 blocks + 32 count]. The state vector allows the AP to context switch between two independent executions much like the register save/restore that allows tasks to context switch on traditional CPU architectures [20, 80]. The output reporting events also encapsulate a flow identifier.

In our architecture, each enumeration path is mapped to an independent flow and time division multiplexed on the same half-core. By association to a flow identifier, we can easily track the enumeration paths that belong to each flow. The host CPU keeps a flow table which tracks the mapping between start states (or enumeration paths) and flows. Each input segment comprising of several flows is processed in several Time Division Multiplexing (TDM) steps. Each flow processes k symbols before a context switch. Once all flows process k symbols, a TDM step is finished. Each TDM step thus processes k input symbols across all flows. The input symbols need to buffered until an entire TDM step is completed. A pointer in the input buffer is rewound to the correct position after each context switch within a TDM step.

The context switch between flows in our system is as fast as 3 AP symbol cycles. This follows from the fact that in our architecture, each enumerated path (and hence each flow) utilizes the same

FSM. Thus there is no need to load the memory arrays or configure the routing matrix during a context switch between flows. To change flows, AP transfers the current state to the state vector cache in the first cycle, then retrieves a previous state from the cache in the second cycle and finally loads it into the mask register (state-enable bits) and counters in the third cycle.

3.2.3 Merging Flows

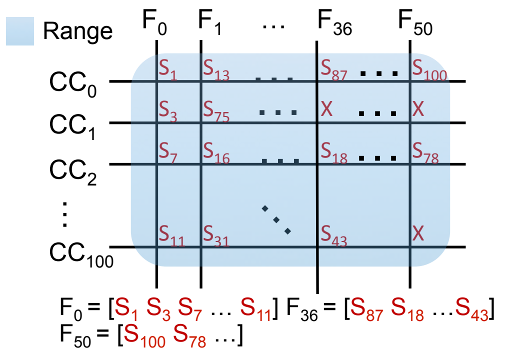
The speedup which can be obtained from our parallelization techniques relies on two factors, the number of input segments executing in parallel and the time taken to complete each input segment. In general, the speedup obtained is equal to number of input segments divided by the slowdown experienced by the slowest input segment. Slowdown of an input segment is simply the time it takes when compared to the time it would have taken had we known the exact start states for that segment.

By utilizing flows, we have maximized the number of input segments. However, time division multiplexing of flows also slows down processing of each input segment. Specifically the processing time of each input segment is proportional to number of flows for that segment. The range guided partitioning method significantly reduces the number of enumeration paths and hence the number of flows needed. However, the number of flows remaining is still large. This section discusses several techniques to further reduce the number of active flows by merging flows.

3.2.3.A Leveraging Connected Components

Intuitively, any two flows can be merged if we can guarantee that there would be no overlap between their state-spaces on any transition, i.e., they belong to different connected components. Since the AP supports any number of simultaneous transitions on a given cycle (subject to routing constraints), we can merge states belonging to different flows and execute them *simultaneously* in the same flow. This observation can be generalized to merge any number of flows as long as we can guarantee their state-spaces do not overlap.

Interestingly we find that many of our benchmarks have a large number of connected compo-



Range
F_0
F_1
…
F_{36}
F_{50}
CC_0
CC_1
CC_2
⋮
CC_{100}
S_1 S_{13} S_{87} S_{100}
S_3 S_{75} X X
S_7 S_{16} S_{18} S_{78}
S_{11} S_{31} S_{43} X
$F_0 = [S_1\ S_3\ S_7 \dots S_{11}]$ $F_{36} = [S_{87}\ S_{18} \dots S_{43}]$
$F_{50} = [S_{100}\ S_{78} \dots]$

Figure 3.3: Merging paths belonging to separate connected components.
Initially we start with the entire range indicated by all the states in the shaded box. Horizontal lines are all states in the range which belong to connected component CC_i. Note 'X' means there is no state. Each vertical line indicates a flow F_j after merging.

nents (or sub-graphs) which do not overlap with each other, i.e., there is no transition edge between the states belonging to different connected components. Intuitively this makes sense because each NFA collects a number of regular expressions or patterns. Patterns with common prefixes belong to the same connected component and patterns which do not share any common prefix belong to separate connected components. Table 3.1 lists the number of connected components in our benchmarks. Our insight is that we can merge flows belonging to separate connected components to reduce the number of flows. The AP compiler in one of its initial stages also partitions the FSM into distinct sub-graphs, however, to ease placement and routing [18]. Figure 3.3 shows our algorithm to merge flows belonging to separate connected components. We group the states obtained by range-guided input partitioning into different connected components. The range table consists of all these states as shown in the shaded box. All the 5000 states in the range are grouped into CC_i groups in the figure. The states on a vertical line through each group were previously mapped to separate flows. Note that we split the states in the same connected component across separate flows so that we can uniquely distinguish them (true paths vs false paths). It can be seen that the number of active flows is equivalent to the number of vertical lines. In the figure we have 50 vertical lines, hence 50 active flows. Thus, we started from 5000 enumeration paths in the range and merged them into 50 flows. Once the flow finishes, the end states of each state belonging to the flow can be uniquely identified by simply masking with a bitmap consisting of the state space of each connected component.

3.2.3.B Active State Group and Common Parent

NFAs usually have several states which are always active due to self-loops on all possible symbols (self-loop is labelled *). These states artificially increase the number of enumeration paths. Given that these states are always active, by definition they belong to the *true path* and can be all combined into one flow which we refer to as the Active State Group (ASG) flow. The output results of this flow are always reported.

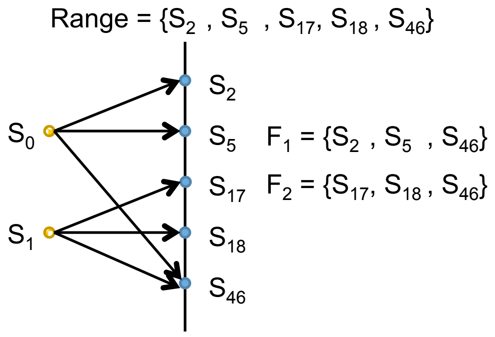


Figure 3.4: Merging states in the range with common parent.

We also observe that states in the range of an input symbol which originate due to the same parent state belong to the same enumeration path. This follows from the fact that in an NFA, there can be many outgoing transitions from a state on a given input symbol. Had we started the input segment one symbol earlier, all these states would have been part of the same enumeration path. Thus, we map all enumeration paths with a common parent to the same flow. Figure 3.4 illustrates the concept. The range consists of states: S_2, S_5, S_{17}, S_{18} and S_{46}. Initially, this would lead to 5 flows. Since S_2, S_5, S_{46} have a common parent S_0, they can be merged into one flow. Similarly, S_{17}, S_{18}, S_{46} have a common parent S_1 and can be merged into one flow, resulting in only 2 flows. Note that for correctness S_{46} has to be included in both flows.

3.2.3.C Dynamic Convergence Checks

Enumerations can be made more efficient by leveraging convergence. We can observe an example of convergence in Figure 3.1. Consider input segment I_2, starting with three different start states S_0, S_1, and S_2. The figure shows three enumeration paths for the state sequence, one for each starting state. After processing the first two symbols, the first two paths get into the same state S_1. After that, these two paths would keep producing the same state sequence as they will observe the same symbols. Hence, there is no need to do redundant computation, and the first two paths can be merged into one path. Thus, an enumeration which started with 3 paths reduces to 2 paths after processing the first two symbols. Prior work on parallelizing DFAs have observed that the state convergence property widely exists in many FSMs [136].

Flow based enumeration allows for easy convergence checks. In our architecture, convergence checks can be implemented by comparing the state vectors in the state vector cache. Comparison requires a simple bitwise logic comparator (one `xor` gate per state bit and a common wired `and`) to be augmented to the state vector cache. Accessing a state vector entry and comparing it to a stored vector takes one symbol cycle. If we have f active flows, convergence checks over all the flows can take up to $f \times f$ symbol cycles. Fortunately, the convergence checks can be entirely overlapped with symbol processing because the state vector cache is not used while processing symbols. However, combining enumeration paths from different connected components into the same flow reduces the probability of convergence. Thus, we invoke convergence checks every ten TDM steps.

3.2.3.D Deactivation Checks

Often many paths in an FSM are not productive. For instance, an enumeration path may process a few symbols successfully, making transitions for each symbol until it comes across a symbol for which none of the active states match. In this case, the path is no longer productive and must be deactivated to save time. In practice, we find many enumerations paths become unproductive after processing a few input symbols. If all the enumeration paths mapped to a flow are unproductive the entire flow can be deactivated to save time. We implement the flow deactivation logic by simply comparing the state bits in the state vector entry to a zero mask during a context switch and invalidate the state vector if there is a match to the zero mask. We observe that many flows get deactivated after processing a few symbols (less than 20 symbols), so we do a few extra deactivation checks even before the first TDM step completes.

3.2.4 Composition of Input Partitions

Once the input segments finish, the final output results can be obtained by combining the results of the true paths of each segment. The host CPU reads the final state vector from the AP and then constructs a Boolean array indicating which flow has results for true enumeration paths. This Boolean array is checked when reading the results out of the output buffer. Each output buffer entry has few bits indicating the flow identifier. *Only results for the output buffer entries which correspond to the true flows are reported.* This computation is done by utilizing the pre-computed range table and masks for connected components (discussed in Sections 3.2.1 and 3.2.3.A).

It takes 1668 symbol cycles to transfer the final state vector from AP to the save buffer of the host CPU [18]. It takes another few tens of symbol cycles to interpret the state vector to figure out which flows encompass true paths in the host CPU. We find that this overhead is not insignificant and thus explore methods to overlap this overhead with input segment processing time on the AP. The *asymmetric finish times of input segments* can be leveraged for this purpose. In general, the different input segments finish at different times based on the different rates of deactivation and convergence. Furthermore, the first input segment executes only the true path, so it is likely to finish quite ahead of the others.

Thus, the first input segment can read its final state vector and create a Boolean array indicating true flows, while the second input segment is still processing its symbols. Thus, its composition overhead is overlapped with the execution of the second input segment. In addition to this, the Boolean array can be utilized to create a Flow Invalidation Vector (FIV) which can be used to invalidate all false flows in the second input segment. In addition to overlapping composition over-

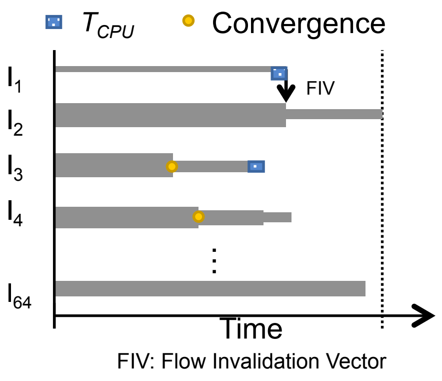


Figure 3.5: Overlapping T_{cpu} with symbol processing.
T_{cpu} can be overlapped with the next segment processing time. The figure also shows the convergence of flows and how the flow invalidation vector (FIV) is used to invalidate unpromising flows.

head whenever possible, this method can further reduce the number of active flows and further speed up input segment processing. The concept can be generalized to all input segments. Figure 3.5 illustrates the above concepts. The first input segment I_1 has only one flow and completes first. The second input segment I_2 on the other hand has many flows (indicated by the thickness of line) and is chugging along. The first segment takes T_{cpu} time to compute its Boolean array for true flows and FIV. The FIV is passed along to input segment I_2. Note that T_{cpu} is overlapped with I_2 processing and after receiving the FIV, the number of flows in I_2 reduces substantially. In some cases when all flows deactivate or converge to only one flow, there is no need to spend T_{cpu} cycles.

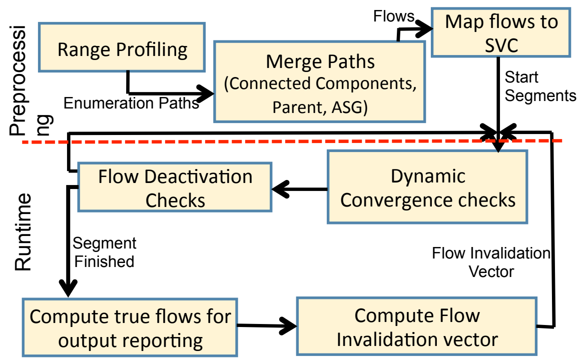


Figure 3.6: Overall framework for low-cost enumeration on the Automata Processor.

3.2.5 Put It Together

This section describes our overall framework for parallelizing NFAs on the AP. Figure 3.6 brings together all the concepts discussed in this section to illustrate our overall framework. The parallelization framework consists of pre-processing steps and dynamic runtime steps. First, the range is computed for all input symbols and a frequently occurring symbol is chosen based on profiling (Section 3.2.1). This is followed by merging the states in the range table into flows based on connected components (Section 3.2.3.A), common parents, and ASG (Section 3.2.3.B). This step generate the contents for the State Vector Cache (SVC). The state vector cache is then loaded onto the AP half-cores. This pre-processing can be augmented to the compilation and configuration process for the AP. Following this, the input is partitioned at boundaries of the chosen range symbol. Each input segment starts getting processed in parallel on the AP half-cores. Deactivation and convergence checks occur dynamically to invalidate redundant or unproductive flows (Sections 3.2.3.D and 3.2.3.C). A segment can also receive a flow invalidation vector from the previous segment during its runtime. Once an input segment finishes, the composition of output reports happens in the host CPU (Section 3.2.4).

3.3 Evaluation Methodology

The proposed approach and optimizations are evaluated on a wide range of benchmark FSMs from the *ANMLZoo* [166] and the *Regex* [43] benchmark suites. These real world benchmarks span multiple domains including network packet monitoring [43], gene sequence matching [148] and natural language processing [191]. Table 3.1 summarizes some of the important characteristics of these FSMs and the parameters used in our simulations. We first describe these workloads in detail, our modifications to these workloads, followed by a discussion on the experimental setup.

#	Benchmark	States	Range	Connected Components	Num. Half-Cores	Input Segments (1 Rank)	Input Segments (4 Ranks)
1	Dotstar03	11124	163	56	1	16	64
2	Dotstar06	11598	315	54	1	16	64
3	Dotstar09	11229	314	51	1	16	64
4	Ranges05	11596	1	63	1	16	64
5	Ranges1	11418	1	57	1	16	64
6	ExactMatch	11270	1	53	1	16	64
7	Bro217	1893	6	59	1	16	64
8	TCP	13834	550	57	1	16	64
9	PowerEN1	12195	466	62	1	16	64
10	Fermi	40783	30027	2399	2	8	32
11	RandomForest	33220	1616	1661	2	8	32
12	SPM	100500	20100	5025	2	8	32
13	Dotstar	38951	600	90	2	8	32
14	Hamming	11254	8151	49	2	8	32
15	Protomata	38251	667	513	2	8	32
16	Levenshtein	2660	2090	4	3	5	21
17	EntityResolution	5689	1515	5	3	5	21
18	Snort	34480	792	90	3	5	21
19	ClamAV	49538	5452	515	3	5	21

Table 3.1: Regex and ANMLZoo benchmark characteristics and AP resources needed.

3.3.1 Workloads

The *Regex* suite consisting of 8 workloads, contains both real-world and synthetic regular expressions primarily meant for network intrusion detection. *ExactMatch* looks for exact pattern matches in the input stream. The *Dotstar* rulesets are parameterized by the fraction of unbounded repetitions of the wildcard $.*$. The *Ranges* dataset accounts for character classes in regular expressions. These are parameterized by the fraction of the ruleset that contains character classes. *Bro217* is an open-source set of 217 regular expressions used for packet sniffing. The *TCP* workload consists of regular expressions used for packet header filtering prior to actual packet inspection.

We use the synthetic trace generator tool from Becchi and others [43] to generate input traces for these workloads. We use traces with $p_m = 0.75$, which is the probability that a state matches on an input character and activates subsequent states as in a depth-wise traversal. $p_m = 0.75$ has been shown to be representative of real-world traffic [43]. Both 1 MB and 10 MB traces are used in our evaluation.

While the *Regex* suite targeted only the network security domain, several recent efforts have uncovered relatively diverse automata-based applications in bioinformatics, data mining and natural language processing that are not necessarily derived from regular expressions [169, 148, 191]. The *ANMLZoo* benchmark suite is one of the first attempts to group these benchmarks and create ”standard candles” for comparing different automata architectures and algorithms. While these benchmarks were developed aiming to saturate the resources on one AP chip, newer versions of the AP compiler place and route some of these automata (e.g., Levenshtein, Entity Resolution) on multiple AP dies since several of these benchmarks are densely connected. We account for this physical automata distribution in our experimental results. The *ANMLZoo* benchmarks along with their input parameters are tabulated in Table 3.1.

The *Snort* ruleset for network intrusion detection is from Snapshot 2.9.7.0. *ClamAV* contains a set of regular expressions from an open source virus database. *Dotstar* in this suite contains a combination of 5%, 10% and 20% wildcard .* repetitions. *Levenshtein* implements the Levenshtein automata used for fuzzy string matching with deletions and insertions allowed. In this suite, strings are of length 24, with edit distance = 3. It is used to match against encoded DNA sequences. *Hamming* is similar to *Levenshtein* and counts the number of mismatches between input strings. *Entity Resolution* has applicability in databases, when the same entity represented with small differences is required to be resolved correctly. For example, names of individuals J. L. Doe and John Doe. *PowerEN* is part of a proprietary set of regular expressions from IBM. *Fermi* predicts high-energy particle paths by matching against known trajectories. *Random Forest* is a machine learning application that implements hand-written digit classification and *SPM* is a data-mining application that mines sequential relations between item transactions to predict future transactions.

It can be seen from Table 3.1 that the state-space of these benchmarks varies greatly and so does the average active set. Furthermore several of these benchmarks also exhibit potential for compression. Similar to the work in [166], we compress automata using the *common prefix merging* technique [42] prior to execution to remove redundant traversals from the automata. For *ClamAV*, *Fermi* and *Random Forest* we do not employ *common prefix merging* as it reduces the number of connected components with only minor benefits in terms of reduction in the number of states. We use both the 1 MB and 10 MB representative input traces provided with each benchmark to evaluate our optimizations. The cost for pre-processing the input stream and post-processing the output reports is minor. Few symbols at the boundary of input segments (64kB for 1 rank and 16kB for 4 ranks) are compared to pre-chosen low-range symbols and chosen for partitioning.

3.3.2 Experimental Setup

We utilize the open-source virtual automata simulator *VASim* [166] to simulate the proposed architecture as well as implement the range-guided input partitioning and all flow merging optimizations discussed in Section 3.2.3. *VASim* allows for fast non-deterministic finite automata emulation by traversing paths only for active states. It supports multi-threading and can partition the input stream and automata processing across many threads. We partition the input stream into nearly equal chunks at symbols with small range and execute a *VASim* context for each flow. Deactivation and convergence of flows is tracked in the simulator as described in Section 3.2.3.

The Automata Processor can deterministically process 1 symbol every 7.5 ns (as long as its output buffers to convey reports are not full), so the latency for symbol processing is known apriori. Context switching between flows requires writing out the old context into the State Vector Cache, reading the new context and loading the new state in the counters and STEs. This has been estimated as 3 cycles [20, 80]. Transferring the 59,936 bit state-vector to the CPU for dynamic invalidation of incorrect flows takes 1668 symbol cycles [19]. On the return path from the CPU, transferring the 512 bit-vector to invalidate flows takes 15 AP cycles. We find that in several of our benchmarks, flows are deactivated before the completion of execution of the previous chunk and we do not incur this extra invalidation overhead in the common case. We assume a latency of 7.5 ns (1 symbol cycle) to determine if any two flows have converged.

We estimate the time taken to identify false paths (and false flows) on a Xeon E3-1240V5 workstation with 8 cores and 32GB RAM. It is also possible for our enumerative approach to falsely trigger reporting elements in some of its false paths. For each benchmark we also account for the overheads of removing these false positives in the output reports as described in Section 3.2.4.

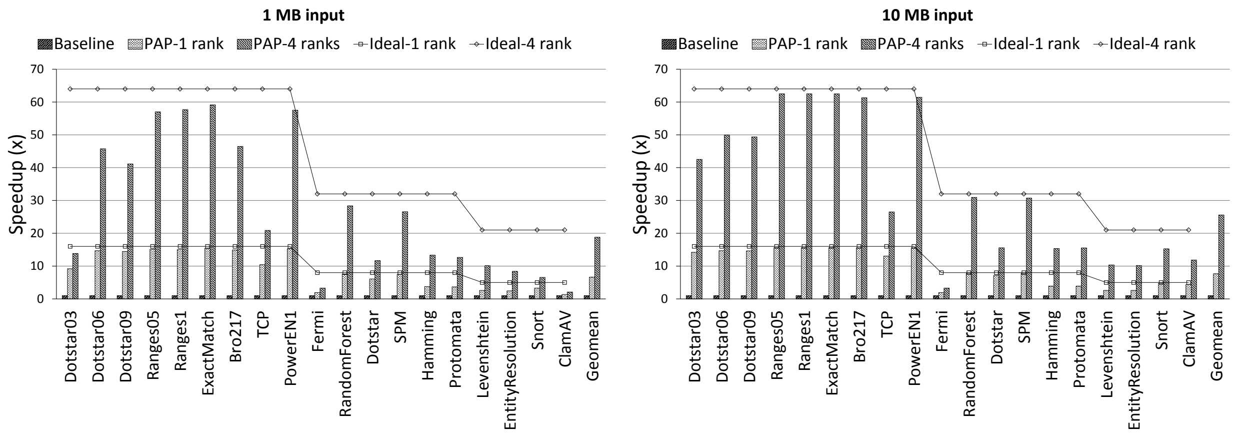


Figure 3.7: Speedup for different inputs.
Baseline: Single AP device. **PAP-1-rank:** One rank with 8 devices of PAP. **PAP-4-rank:** Four ranks with 32 devices of PAP. **Ideal-1-rank:** One ideal AP rank with 8 devices and no parallelization overheads. **Ideal-1-rank:** Four ideal AP ranks with 32 devices and no parallelization overheads.

3.4 Results

In this section we first present the speedups obtained by our proposed architecture, followed by a detailed explanation of the reasons for this speedup. We also present an analysis of the different sources of overhead introduced by the proposed optimizations.

3.4.1 Overall Speedup

Figure 3.7 shows the speedups obtained by our proposed Parallel Automata Processor Architecture (PAP), when compared to the baseline AP architecture. We present speedups for both 1 AP rank (8 D480 devices) and 4 AP ranks (32 D480 devices in the current AP generation) and 1 MB and 10 MB input streams. We also exploit the parallelism offered by each of the half-cores in a D480 device when our FSMs can fit in a single half-core. Table 3.1 details the AP footprint and number of input segments created for each of our benchmark FSMs. The *Ideal* legend in the figure equals the number of input segments that can be processed in parallel. Overall, across the complete range of 19 benchmark FSMs, for the 1 rank and 4 rank cases, PAP achieves 6.6× and 18.8× speedup for the 1 MB input stream and 7.6× and 25.5× speedup for the 10 MB input stream.

It can be seen from the figure that PAP outperforms the sequential AP baseline for most benchmarks. A noticeable trend is the larger performance gains with the 10 MB stream. This is because the larger stream provides opportunities for creating larger input segments. These larger input segments help in reducing in the number of active flows due to the deactivation and convergence properties of the FSMs discussed in Sections 3.2.3.C and 3.2.3.D and the associated flow switching overhead. Furthermore, large input segments also help amortize the cost of false path invalidation and input composition in the CPU. For benchmarks with small input symbol ranges, in particular *Ranges05*, *Ranges1* and *ExactMatch*, PAP achieves near ideal speedup both in the 1 rank and 4 rank cases. Even FSMs with significantly large number of initial flows like *SPM* (20101) and *Hamming* (8152), PAP achieves greater than 16× speedup in the 10 MB case because of the connected components and common parent optimizations for flow reduction discussed in Section 3.2.3. A detailed analysis of our optimizations for flow reduction is presented in the next section.

It is also important to note that the current generation of AP only supports 512 active flows in its State Vector Cache per D480 AP device. Several of the studied FSMs significantly exceed the 512 limit as can be noticed from the `Range` column entries in Table 3.1. The proposed flow reduction optimizations are therefore essential to the success of the proposed parallelization approach. For benchmarks like *Fermi*, consisting of a large number of active states and *Entity Resolution* with highly dense connected components, our optimizations are unable to significantly reduce the initial number of active flows, limiting speedups.

Our parallel approach is never worse than the sequential baseline as the half-core processing the first input segment, after completion, continues to process the remaining segments (golden execution). In case this half-core finishes processing all input segments, we invalidate all other executing flows and report results for the golden execution. A more aggressive policy need not wait for the completion of the golden execution. It can invalidate all the flows after the golden execution has finished x segments (with x calculated based on the minimum expected speedup).

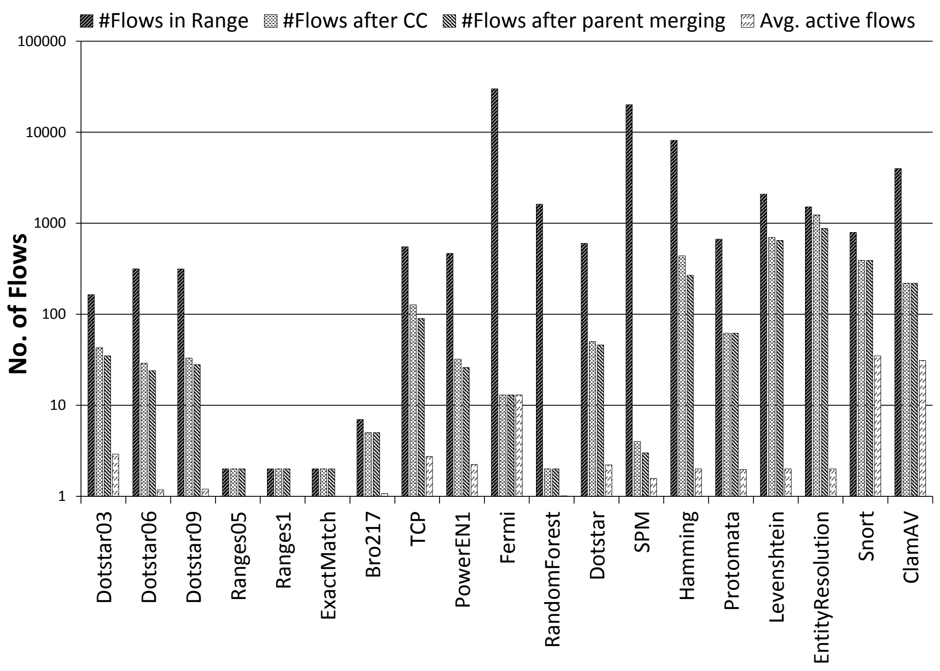


Figure 3.8: Average number of flows across benchmarks.

3.4.2 Active Flow Set

Figure 3.8 shows the contribution of each of our flow reduction optimizations in achieving the speedups discussed in the previous section. We also plot the average number of active flows in different benchmarks for the 1 MB input stream case as an example. Note that the y-axis scale is logarithmic.

While benchmarks with small input symbol ranges are inherently good candidates for the flow-based enumeration scheme, it can be seen that several of our benchmarks have greater than 1000 states in their initial range. In particular *SPM* consisting of 20101 initial flows and 5025 distinct connected components greatly benefits from the connected components optimization which reduces these to 5 flows. Note that our pre-processing step identifies frequently occurring input symbols with small range. In their absence, other optimizations like connected components prune the number of enumeration flows as discussed above. All optimizations work synergistically to reduce the number of enumeration paths.

We noticed that even though the connected components optimization greatly helped reduce the number of flows (e.g., from 467 to 32 for *PowerEN*) several flows remained active for the complete execution. Investigating further revealed that the connected components optimization artificially creates more flows for states originating from the same parent as discussed in Section 3.2.3.B. With the proposed common parent merging algorithm, we achieved a 1.6× and 1.4× reduction in flows for *Levenshtein* and *Hamming*. Also, the dynamic flow convergence and deactivation checks discussed in Section 3.2.3 contribute to a great reduction in number of active flows for all benchmarks. We see an order of magnitude reduction in number of flows for *Dotstar0x* and several orders of magnitude improvement for *RandomForest*, *Fermi* and *SPM*.

3.4.3 Overheads

This section discusses the different sources of overhead in the proposed PAP architecture.

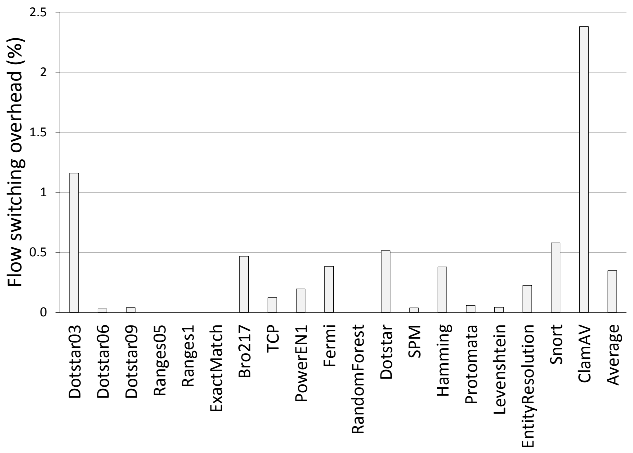


Figure 3.9: Costs of flow switching.

Flow Switching and Dynamic Checks: It can be seen from Figure 3.9 that the overheads of context switching between flows are less than 2% for most benchmarks. As discussed before, since the number of active flows greatly reduces as input symbols are processed, the corresponding convergence and deactivation checking overheads also reduce. Furthermore, these checks can be overlapped with symbol processing. *ClamAV* however has a large number of active flows and sees 2.4% overhead. This accounts for the relatively low speedup for *ClamAV* when compared to other benchmarks in Figure 3.7.

We also simulated the effect of larger context switch times. Our speedup for 1 MB inputs reduces by on average, 0.5% (1.75% worst case) and 1.2% (5.04% worst case) for 2× (6 cycles) and 4× (12 cycles) context switch time respectively. The context switch overhead is proportional to the number of active flows, which greatly reduce as symbols are processed due to convergence and deactivation. Also, dynamic convergence checks can be overlapped with symbol processing, since these checks are carried out on state vector cache entries, which do not participate in symbol processing (state transitions).

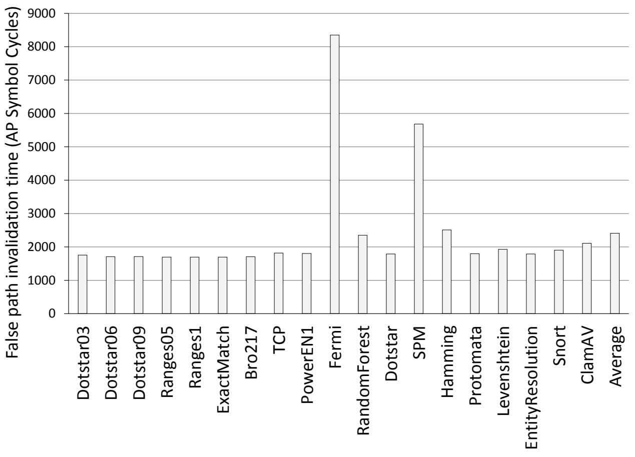


Figure 3.10: False path decoding costs

False Path Decoding: Figure 3.10 illustrates the overheads of decoding false paths at the host CPU after an input segment finishes and sending a flow invalidation vector (FIV) to the next segment. On average, most benchmarks see around 2000 symbol cycles overhead. Fortunately, this cost is largely amortized because of two reasons: (1) it can be overlapped with symbol processing in subsequent segments, (2) these invalidations are infrequent since several flows have either already converged or have been deactivated and do not require this invalidation.

Output Reports: The AP uses reporting elements to inform the host CPU about pattern matches against the input. The host reads the output event buffer on the AP and decodes each entry to finally report matches to the user. Since our approach uses enumeration for parallelization, *false paths* are traversed and output events may be generated along these false paths. Figure 3.11 illustrates the increase in output reports due to false paths for each of the benchmarks. These false

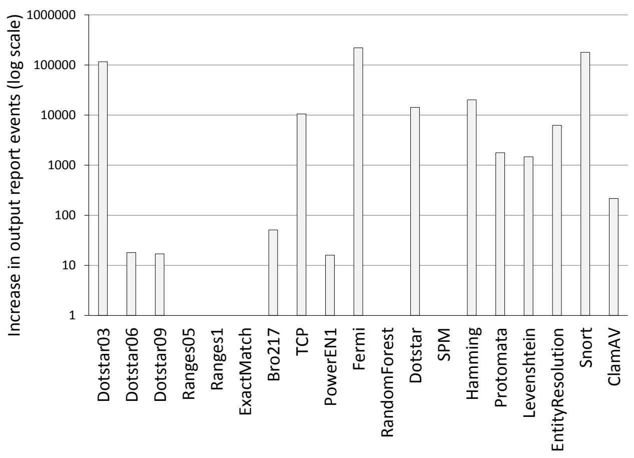


Figure 3.11: Increase in output reports due to false paths

positives are filtered out on the host. We account for the time taken for post-processing the output reports in both baseline AP and PAP for our final speedup measurements shown in Figure 3.7.

On average, output reporting and worst case FIV on host CPU take $\sim$1% and $\sim$6% of total execution time respectively, without accounting for the overlap of FIV computation. This is because output reporting and flow invalidation are infrequent.

Energy: Since the PAP architecture reduces overall execution time, we expect a reduction in static energy. However, in the PAP architecture, we activate more state-transition-elements than the baseline, arising from the traversal of false paths which can lead to an increase in dynamic energy. On average, there are 2.4$\times$ extra transitions per input symbol. State activation only writes to multiple flip-flops (mask register) and does not require additional writes to the DRAM or activation of a large number of additional DRAM rows. The AP activates an entire DRAM row for every input symbol and reads out different columns based on the bits stored in these flip-flops. Therefore, these additional activations do not lead to significantly increased dynamic energy costs.

3.5 Summary

This paper attempts to break the sequential NFA execution bottleneck on the Micron Automata Procesor (AP). We identify two main challenges to applying enumerative NFA parallelization techniques on the AP: (1) high state-tracking overhead for input composition (2) huge computational complexity for enumerating parallel paths on large NFAs. Using the AP flow abstraction and properties of FSMs like small input symbol transition range, connected components and common parents we amortize the overhead of state-tracking and realize a time-mutliplexed execution of enumerated paths. To tackle the computational complexity, we leverage properties of FSMs like path convergence to dynamically reduce the number of executed enumerated flows. The algorithmic insights about large real-word NFA provided in this work (e.g., presence of connected components, common parents, active state groups, range partitioning) are general and can be applied to parallelize NFA execution on any spatial, data-flow substrate with memory and interconnects, like FPGAs, cache sub-arrays or memristor crossbar arrays. What is required is an efficient state-encoding and state-mapping scheme along with a mechanism for supporting state-transitions using interconnects. Furthermore, different connected components and flow contexts may also be mapped to separate GPU threads for parallelism. Our evaluation on a range of FSM benchmarks shows 25.5$\times$ speedup over the sequential baseline AP with 64 half-cores.

CHAPTER 4

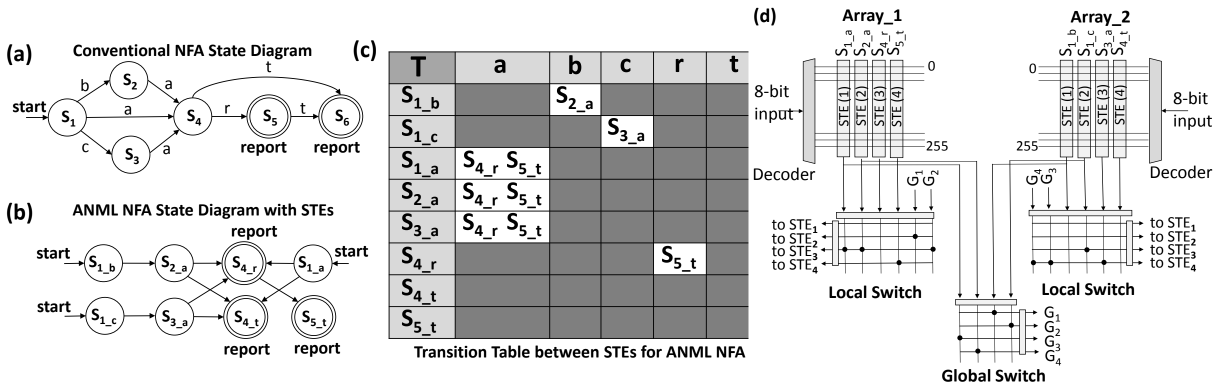
Cache as a Pattern Matching Engine

While memory-centric architectures such as Micron's Automata Processor have been highly successful in accelerating automata processing, implementing the state-transition logic on top of lower process node DRAM has significant area overhead (upto 16.6$\times$). In this chapter, we explore the possibility of repurposing the last-level cache of modern processors to accelerate automata processing. We discuss the benefits of this approach, the associated challenges involved and propose the Cache Automaton architecture to enable in-situ automata processing in the last-level cache.

4.1 Cache Automaton Concept

ANML NFA computation entails processing a stream of input symbols one at a time. Initially, all the start states are active states. Each step has two phases. In the *state match* phase, we identify which of the active states have the same label as the current input symbol. In the next *state transition* phase, we look up the transition table to determine the destination states for these matched states. These destination states would become the active states for the next step. Now, we discuss how Cache Automaton implements these two phases efficiently.

State Match: We adapt Micron's AP processor [64] design for implementing the state match phase. Each NFA state is mapped as a State Transition Element (STE) to a *column* of an SRAM array in the last-level cache. The value of an STE column is set to the one-hot encoding of the 8-bit input symbol it is mapped to. This means that each STE (or column) is 256 bits and each bit position signifies an input symbol in the ASCII alphabet. Figure 4.2 (a) shows an SRAM array in



T	a	b	c	r	t
S_{1_b}		S_{2_a}			
S_{1_c}			S_{3_a}		
S_{1_a}	S_{4_r} S_{5_t}				
S_{2_a}	S_{4_r} S_{5_t}				
S_{3_a}	S_{4_r} S_{5_t}				
S_{4_r}				S_{5_t}	
S_{4_t}					
S_{5_t}					

Figure 4.1: Example NFA mapping to SRAM arrays and switches.
An example NFA and its mapping to two small SRAM arrays and switches. The NFA accepts patterns {`bat, bar, bart, ar, at, art, car, cat, cart`}.

Cache Automaton which holds 256 STEs. Every cycle, the current input symbol is broadcasted to all SRAM arrays as a *row address* and the corresponding row is read out. If an STE has a '1' bit set in the row, it means that the label of the state it has stored matches the input symbol. Thus, by broadcasting the input symbol to all SRAM arrays, it is possible to determine in parallel all the states which *match* with the current input symbol.

The row corresponding to the input symbol is read out and stored in a match vector. An active state vector (one bit per STE; 256-bit vector in our example) stores which STEs are active in a given cycle. A logical AND of the match and active state vectors determines the subset of active states which match the current input symbol. The destination states of these matched states would become the next set of active states. Section 4.2.3 discusses solutions to implement this state match phase efficiently in cache sub-arrays.

State Transition: This phase determines the destination states for the matched states found in the previous phase. These states would then become the next set of active states. We observe that a matrix-based crossbar switch (essentially a $N \times N$ matrix of input and output ports) is suitable to encode a transition function. In a crossbar, an input port is connected to an output port via a cross-point. Each STE connects to one input port of the switch. A cross-point is enabled if the input STE connects to a specific output STE. The result of state-matches serve as inputs to the switch, and the output of the switch is the next set of active states.

The switches in the cache automaton architecture have modest wiring requirements (256-512 input and output wires; see Table 4.2), as the data-bus width is only 1-bit. However, cache automaton switches have two major differences from conventional switches. *First*, there is no need for arbitration. The connections between the input and output ports can be configured once during initialization for an NFA and then be used for processing all the input symbols. Since there is no arbitration, the enable bits must be stored in the cross-points. Automaton switches have a large number of cross-points, and therefore we need a compact design to store the enable bit at each cross-point. *Second*, unlike a conventional crossbar, an output can be connected to multiple inputs at the same time. The output is a logical OR of all active inputs. Section 4.2.4 discusses our proposed switch architecture for Cache Automaton which supports the above features.

Ideally, the entire transition function could be encoded in one switch to provide maximum connectivity. However, such a design will be incredibly slow. To scale to thousands of states and many SRAM arrays, we adopt a hierarchical switch architecture as discussed in Section 4.2.1.

4.1.1 Working Example

We describe a simplified example which brings together all the above concepts. Figure 4.1 shows an example NFA which accepts patterns {`bat, bar, bart, ar, at, art, car, cat, cart`} and how it is mapped to SRAM arrays and switches. The figure starts with a classical representation of an NFA in terms of states and transitions (Figure 4.1 (a)). Figure 4.1 (b) shows the ANML NFA representation for the same automata. Figure 4.1 (c) shows the transition table for the ANML NFA with STEs. This example NFA requires only 8 STEs. Real world NFAs have tens of thousands of states which need to be mapped into hundreds of SRAM arrays.

For this example, let us assume we have two small SRAM arrays which can each accommodate 4 STEs as shown in Figure 4.1 (d). The NFA requires 8 STEs, so we split the states into 4 STEs in Array_1 and Array_2. Each array has a 6×4 local switch, and together they share a 2×2 global switch. Each STE can connect to all STEs in its array. In this example, only two STEs (STE_1 and STE_2) in an array are allowed to connect to all STEs in the other array via the global switch.

The transition table in Figure 4.1 (c) is mapped to local and global switches. For instance, S_{1_a} and S_{4_r}, are mapped to STE_1 and STE_3, of Array_1. Since S_{1_a} can transition to S_{4_r}, the local switch cross-point between STE_1 and STE_3 is set to connected (represented by black dot). The figure also shows how a connection via the global switch is established for states S_{2_a} mapped to STE_2 of Array_1, and S_{4_t} mapped to STE_4 of Array_2. This is accomplished by: (1) feeding STE_2 as an input to the global switch, (2) connecting the second input of the global switch to the G_4 output which feeds as an input to Array_2's local switch, (3) G_4 input is connected to the STE_4 output (or S_{4_t}) of Array_2's local switch.

4.2 Cache Automaton Architecture

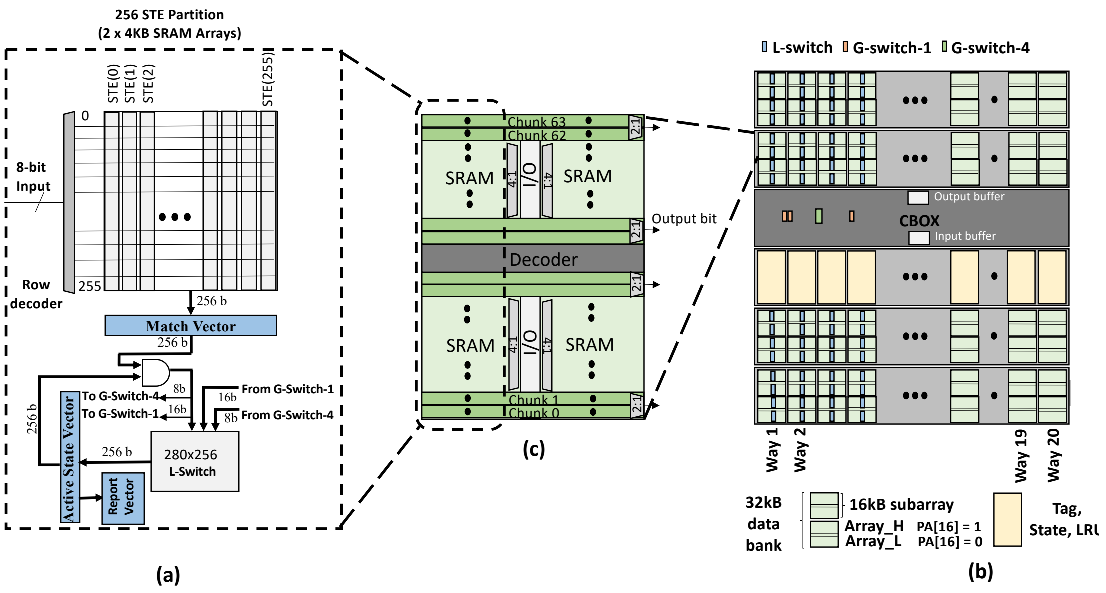


Figure 4.2: Cache Automaton Architecture.
The figure shows (a) SRAM arrays repurposed to store 256 STEs, (b) one 2.5 MB Last-Level Cache (LLC) slice architecture and (c) Internal organization of one 8 KB sub-array.

4.2.1 Cache Slice Design

The proposed cache automaton is implemented in the Last-Level Cache (LLC) in order to accommodate large NFA with thousands of states. Figure 4.2 (b) shows the overall organization of a slice of LLC with the Cache Automaton architecture. The depicted LLC slice is modelled *exactly* after Xeon E5 processors [90, 55]. Each LLC slice is 2.5 MB. Intel processors support 8-16 such slices [46]. Each slice has a central cache control box (CBOX). Remainder of the slice is organized into 20 columns. A column consists of eight 16 KB data sub-arrays, and a tag array. Each column represents a way of set-associative cache. Internally a 16 KB data sub-array consists of *four* SRAM arrays with 256×128 6T bit-cells as shown in Figure 4.2 (c). Each array has 2 redundant columns and 4 redundant rows in the SRAM arrays to map out dead lines. STEs are stored in the columns of the 256×128 SRAM array. A 16 KB data sub-array can store up to 512 STEs. We define a *partition* as a group of 256 STEs mapped to two SRAM arrays each of size 4 KB.

Our interconnect design is based on the observation that real-world NFA states can typically be grouped into partitions, with states within a partition requiring rich connectivity and states in different partitions needing only sparse connectivity (details in Sections 4.3.1 and 4.3.2). To support this partitioning, we add one local switch (*L-Switch*) per partition providing rich intra-partition connectivity and global switches (*G-Switch*) for sparse connections between partitions across a way or multiple ways. Each L-Switch is of size 280×256, i.e., 280 input wires and 256 output wires. The input wires correspond to 256 STEs in the sub-array and 16 input wires from the G-Switch in the same way (*G-switch-1*) and 8 input wires from the G-switch connecting the 4 ways (*G-switch-4*). A STE in a partition can connect to any other STE in its partition via the L-Switch. Also, 16 STEs from a partition can connect to other partitions in the same way via *G-Switch-1* and 8 STEs from a partition can connect to other partitions via *G-switch-4*. We also observe that even without connections between global switches, the proposed interconnect topology provides sufficient headroom to our compiler to map all the evaluated NFA.

Figure 4.2 (a) shows a single partition. The input symbol match result is read out and stored in the *match vector*. The logical AND result of the match vector and the active state vector is fed as an input to the local-switch (256 STEs) and the global-switches (16 STEs and 8 STEs respectively). After the signals return from the global-switches to the local-switch, the next active state vector is available as the output of the local-switch. This is written back to the active-state vector. If any of the final or reporting states are active, then an entry is made into the output buffer in the CBOX recording the match (Section 4.2.5).

4.2.2 Automaton Pipeline

The automaton processes a stream of input symbols *sequentially* and hence the time to process each input symbol determines the clock period. The clock period determines the rate of processing input symbols and hence the overall system performance. We observe that each input symbol is processed in two independent phases, SRAM access for state match and propagation through the interconnect (switches and wires) for state transition. Furthermore, SRAM access for the current input symbol can be overlapped with the interconnect delay for processing the previous input symbol. Based on this observation, we design a three-stage pipeline for *Cache Automaton*. Typical application scenarios process large amounts of input data (MBs to GBs), thus the pipeline fill-up and drain time are inconsequential.

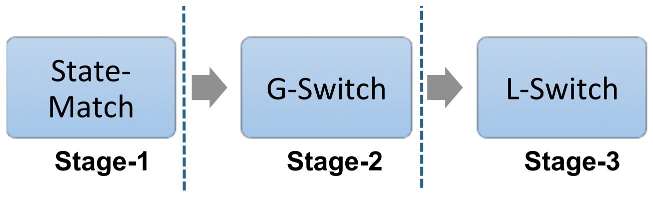


Figure 4.3: Three-stage pipeline design for Cache Automaton.

The pipeline stages are shown in Figure 4.3. The first stage of the pipeline is state-match or SRAM array read access. The output of this stage is stored in the match vector (Figure 4.2 (a)). It is moved to another buffer at the end of the stage to make room for the next state match. The second stage of the pipeline is propagation through the global switch (G-Switch). This includes the wire delay from the SRAM array to the global-switch. The output of this stage is stored in the output latches of the G-Switch. The third and last stage of the pipeline is propagation through the local switch (L-Switch). This includes the wire delay from G-Switch to L-Switch. This stage writes the next states to the active state vector and completes processing of the current input symbol. Note, the output of the L-switch updates the active state vector, and the active state cannot be updated until transition signals from other partitions have reached their destination L-switches via the G-switch. Thus, G-Switch forms the second stage of pipeline followed by L-switch.

As can be inferred from the above discussion, the symbol processing time for *Cache Automaton* is determined by symbol-match delay and switch delay. Therefore, for high-performance, it is critical to speed up both the symbol match delay and interconnect delay. The next two sections discuss techniques towards this end. Section 4.5.1 quantifies the delay of each pipeline stage and the overall operating frequency.

4.2.3 Enabling Low-Latency Matches

The input symbol match for cache automaton is simply an SRAM array read operation. Conventional SRAM arrays share I/O or sense-amplifiers for increased packing density. This results in column-multiplexing which increases the input symbol match time significantly. For example, a 4-way column multiplexed SRAM array shares one sense-amplifier across four bit-lines and hence can read out only 1 bit out of 4 bits in a cycle as shown in the left side of Figure 4.4. A read (or match operation) consists of decoding, bit-line precharging (PCH), and I/O or sensing as shown in the baseline timing on the right side of Figure 4.4. A 4-way column multiplexed array requires 4 cycles to read out all the bits in a row of the array.

A dynamic scheme which checks the active state vector and matches fewer STEs, can save energy. Unfortunately, this cannot improve performance because the *clock period is determined by worst case state-match time*. However, we observe that unlike conventional cache read accesses, automata state transitions need to read all the bits which are column multiplexed. This can be leveraged to improve match latency, by cycling the sensing phase. All the bitlines of an SRAM array can be precharged in parallel, followed by sequentially sensing the column multiplexed bits.

Figure 4.4 shows the timeline of an optimized read sequence for 4-way column multiplexing. First precharge (PCH) is asserted, followed by a read word-line (RWL) assertion. By the end of RWL assertion all bi-lines are ready for sensing. The sense-amplifier enable (SAE) is asserted in 4 steps. The column-multiplexer select signal (SEL) is set to 0 to read the first column, and

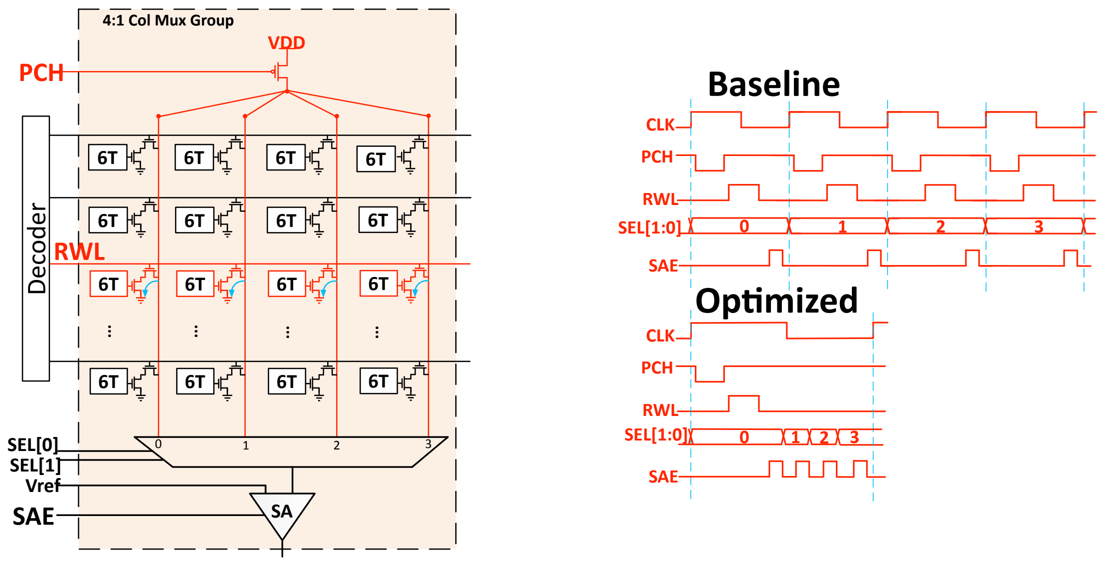


Figure 4.4: Design and timing diagram for a 4-way multiplexed SRAM column.

changes to 1,2, and 3 with each SAE assertion. Typically SRAM arrays use pulse generators for control signals like SAE, PCH and RWL and these pulses are not generated using a separate higher frequency (i.e., 8 GHz) clock. These consist of a chain of high-Vt, long channel, current-starved inverters and NAND gate that can be configured to generate pulses of widths 1/2–1/4 of clock period. In our case, a 125 ps (8 GHz) pulse can be generated for SAE and SEL. Power and area overhead for the pulse generator is minimal – 8-10 inverters switching once every clock cycle (8 μW). Since sensing takes about 25% of the cycle time, this optimized read sequence can read all the bits for a 4-way column multiplexed array 2× faster than the baseline. For 8-way column multiplexing the benefits from this optimization are higher.

4.2.4 Switch Design

This section discusses the proposed switch design. As explained in Section 4.1, an automaton switch needs to support two new features. First, since it has a large number of 1-bit ports, it needs to store a large number of cross-point enable bits. There is no need for dynamic arbitration. But the switch needs to provide a configuration mode which allows writing to the enable bits. Second, to allow efficient many-to-many state transitions, an output needs to be connected to multiple inputs and be the equivalent of logical OR of the active inputs.

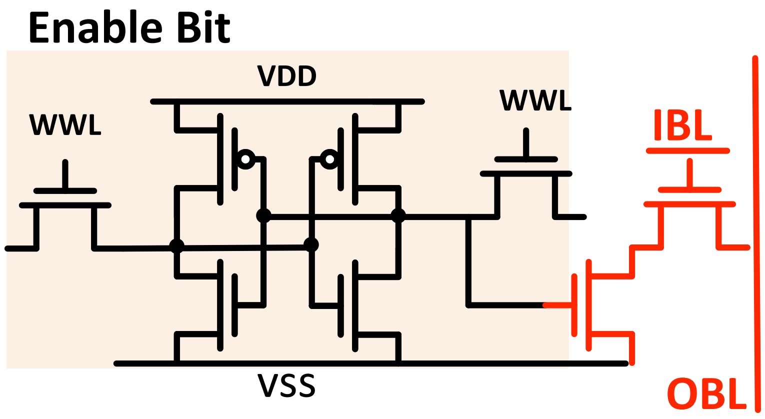


Figure 4.5: The 8T cross-point design for switches.

To support the above two features, we developed an 8 transistor (8T) cross-point design as shown in Figure 4.5. The enable bit which controls the connection of input bitlines (IBL) to output bitline (OBL) is stored in a 6T bitcell. The connection between IBL and OBL is via a 2T block. The switch supports two modes: crossbar mode and write mode. During the *crossbar mode*, the OBLs are pre-charged. If any of the IBLs carry a '0', the OBL is discharged. Thus outputs carry a wired AND of inputs. Note, the inputs and outputs are active low. Thus, the final result on an output wire is the logical OR of all inputs. Each OBL has a dedicated sense-amplifier. During the *write mode*, the 6T enable bits can be programmed by writing to all bitcells sharing one write word-line (WWL) in a cycle. The switch is provisioned with a decoder and wordline drivers for the write mode. The proposed switch can take advantage of standard 8T push rule bit cells to achieve a compact layout.

4.2.5 Input Streaming and Output Reporting

Cache Automaton takes a steady stream of input symbols and produces intermittent output matches. Input symbols (1 byte each) are stored in a small 128 entry FIFO in the C-BOX as shown in Figure 4.2. The FIFO is associated with an input symbol counter which indicates the symbol cycles elapsed. One input symbol is read from the FIFO every cycle and broadcasted to all SRAM banks by using the existing address bus within a cache slice. Our model assumes that applications copy input data to a cache location. In the *Cache Automaton* mode, as input symbols get processed and deleted from the FIFO, the cache controller reads a cache block worth of input data via regular cache access and fills up the FIFO.

We follow a model similar to Micron's AP for output reporting. An output buffer has 64 entries, an entry for each match with a reporting state. An interrupt is sent to the CPU if all the output buffer entries are full. The report states in the NFA can be mapped to designated STEs in a partition. A 256-bit mask indicates if the reporting states are mapped to these STEs. A wired OR of the result obtained from the logical AND of the active state mask and output reporting mask (report vector) triggers an output reporting event. An output reporting event creates a new entry in the output buffer. Each entry consists of the active state mask, partition identifier, input symbol, and offset in the input stream.

4.2.6 System Integration

While repurposing the last-level cache for automata processing, the following system-level issues need to be kept in mind:

Sharing Model with CPU: Our architecture is aware of a *way* in LLC, but there is no mode for directly addressing a cache way in x86 instructions. A load address can be mapped to any way in the LLC. To overcome this limitation, Cache Automaton leverages Intel's Cache Allocation Technology (CAT) [5] to dynamically restrict the ways accessed by a program and thus exactly control which cache way the data gets written to. NFA computation is carried out only in 4-8 ways of each slice. The remaining 12-16 ways can be used by other processes/VMs executing on the same/different cores without leading to starvation in inclusive caches. By associating the NFA process to one of the highest `cgroups` (class-of-service), CAT can ensure that incoming data from processes in low-priority `cgroups` does not evict data in active NFA ways and guarantees QoS during steady-state. With regard to addressing, LLC hashing must be disabled during configuration time to place STE data into specific slices. This can be done by writing to special Model-Specific-

Registers (MSRs) like those used to associate L3 cache ways with `cgroups` in CAT. Note that LLC hashing need not be disabled during normal execution.

Power Management: Since NFA computation has high peak power requirements for some benchmarks, the OS scheduler together with the power governor must ensure that the system TDP is not exceeded while scheduling other processes simultaneously on the CPU cores. Based on the number of cache arrays, ways, slices allocated for NFA computation and average active states for representative inputs, the compiler can provide coarse-grained peak-power estimates (hints) to guide OS scheduling. In case the OS wishes to schedule a higher-priority process, the NFA process may also be suspended and later resumed by recording the number of input symbols processed and the active state vector to memory. It must be noted that while peak power is high, the energy consumed is orders of magnitude lower than that expended by conventional CPUs due to savings in data movement and instruction processing overheads.

4.2.7 Configuration and ISA Interface

We adopt a configuration model similar to Micron's Automaton Processor (AP). Before processing the NFA, the switches have to be configured and the cache arrays have to be initialized with STEs. The switches can be configured by utilizing their write-mode, during which they simply function as SRAM arrays. We assume switch locations are memory-mapped and addressable by CPU load instructions.

The initialization of cache arrays can be done by CPU load instructions which fetch data from memory to caches. Our compiler creates binary pages which consists of STEs stored in the order in which they need to be mapped to cache arrays. The compiler carefully orders the STEs based on the physical address decoding logic of the underlying cache architecture. These binary pages with STEs are loaded in memory, just like code pages. Most LLC cache sets are addressable by last 16 bits of memory address. These bits can be kept same for both virtual pages and physical pages by mapping the binary data (containing STEs) to huge pages [10]. LLC hashing is disabled during configuration as discussed in Section 4.2.6.

The average initialization time is dictated by the number of SRAM arrays occupied by an NFA. For our largest benchmark, we found this to be about $0.2ms$ on a Xeon server workstation. In contrast AP's configuration time can be up to tens of milliseconds [149]. Once configured, in typical use cases such as log processing, network traffic monitoring and DNA motif search, NFAs typically process GBs/TBs of data, thus processing time easily offsets configuration time. But configuration may be costly when frequently switching between structurally different automata. For these, optimizations like overlapping the configuration of one LLC slice with processing in others and prefetching may be explored. We leave this exploration to future work.

An ISA interface is required to specify: (1) when to start processing in Cache Automaton mode and (2) start address from which input data needs to fetched to fill up input FIFO buffer. (3) the number of input symbols to be processed. One new instruction can encapsulate all the above information. Compiler can insert this special instruction in code whenever it needs to process NFA data. An interrupt service routine handles output buffer full reporting events.

4.3 Cache Automaton Compiler

In this section, we explain our compiler that takes as input an NFA description consisting of several thousands of states and efficiently maps them onto cache banks and sub-arrays. Care is taken to ensure maximum cache utilization while respecting the connectivity constraints of the underlying interconnect architecture. The algorithms proposed are general and the insights provided are also applicable for mapping NFAs to any spatial reconfigurable substrate with memory like FPGAs or memristor crossbar arrays.

The compiler takes as input an NFA described in a compact XML-like format (ANML) and generates a bit-stream containing information about the NFA state to cache array mapping and the configuration enable bits to be stored in the various crossbar switches of the automaton interconnect. We propose two mapping policies leading to two *Cache Automaton* designs, one optimized for performance CA_P and the other optimized for space utilization CA_S. Before proceeding to explain the mapping algorithm used by the compiler (Section 4.3.2), we motivate the insights behind each of these designs in Section 4.3.1.

4.3.1 Connectivity Constraints

Real-world NFAs are composed of several *connected components (CCs)* with only a few hundred states each. Each *connected component* describes a pattern or group of common patterns to be matched. Since these *connected components* have no state transitions between them, they can be treated as atomic units by the mapping algorithm. Each connected component can be viewed as operating independently matching against the input symbol in parallel.

Performance Optimized Mapping: In their baseline NFAs, most benchmarks have connected components with less than 256 states (refer Table 4.1), making it possible to fit at least one *connected component* in each partition (256 STEs stored in two 4 KB SRAM arrays, Figure 4.2 (a)). This motivates our *performance-optimized* mapping scheme that greedily packs connected components onto cache arrays. We were able to operate all of the baseline NFA benchmarks while limiting the connectivity across cache arrays to within a way. Only cache arrays which are mapped to physical address with A[16] = 0 (Array_L in Figure 4.2 (c)) are used for mapping NFA and cache arrays with A[16] = 1 (Array_H in Figure 4.2 (c)) can be used for storing regular data provided that compiler can ensure that regular data are placed in 64 KB segments in the virtual address space or the OS does not use physical pages with A[16] = 1. As we discuss in Section 4.5.1, the performance-optimized design can operate at a frequency of 2 GHz.

Space Optimized Mapping: However, just using the baseline NFAs forgoes algorithmic optimizations on NFAs that seek to remove redundant automata states and state traversals. These redundancies are common in practice, since many patterns share *common prefixes* (for example, patterns like art and artifact) and these common prefixes can be matched once for all connected components together. Eliminating redundancies helps reduce the space footprint of the NFA. It also reduces the average number of active states, leading to reduction in dynamic energy consumption. This has been the motivation for several *state-merging algorithms* in literature that merge common prefixes across pattens [42]. However, it must be kept in mind that since these optimizations merge states across many connected components they tend to reduce the number of connected components and *increase the average connected component size*. Larger connected components require richer connectivity and demand more interconnect resources (crossbar switches and wires). To support such state-merged NFA, we propose a *space-optimized* mapping policy that leverages graph partitioning techniques (Section 4.3.2) to minimize outgoing edges between partitions. We also provision additional global crossbar switches to ensure connectivity across 4-8 ways of a cache slice. Our hierarchical switch design provides a richer average fan-out transitions and fan-in transitions per state compared to the AP (Section 4.5.4). However, richer connectivity comes at the cost of higher latency due to increased wire delay, therefore the *space-optimized* design operates at a lower frequency (1.2 GHz) compared to the *performance-optimized* design (2 GHz).

4.3.2 Mapping Algorithm

The algorithm takes as input the ANML NFA description of the benchmark, the number of cache arrays available and the size of each cache array. The output is a mapping of NFA states to cache arrays. It operates in three steps. In the first step, all connected components which have size less than *partition_size* (i.e., 256 states) are identified. As discussed earlier, a connected component forms the smallest mapping unit. Next, these connected components are mapped greedily onto the cache arrays to pack multiple connected components onto the same cache array when possible. We do not partition the connected component in the first stage, since the connected component inherently groups together states that have plenty of state-transitions between them and mapping these states to the same array leads to a more space-efficient packing. Connected components larger than *partition_size*, need to be partitioned across k-different partitions (in the same way or across multiple ways of the cache slice). We utilize the open-source graph partitioning framework *METIS* [97] to solve this *k-way* partitioning problem. *METIS* partitions the connected component into different partitions such that the number of outgoing state transitions between any two partitions is minimized. It works by first coarsening the input connected component, performing bisections

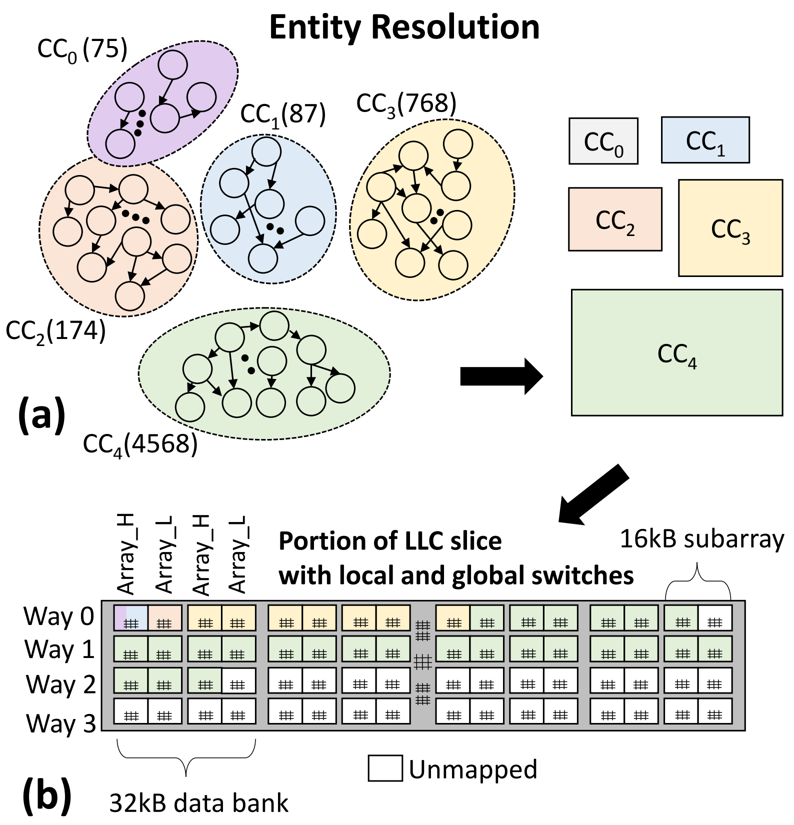


Figure 4.6: Mapping *Entity Resolution* to cache arrays.
Figure showing the mapping of connected components in *Entity Resolution* to cache arrays. The connected components are labeled along their size in brackets.

on the coarsened connected component and later refining the partitions produced to minimize the edge cuts. We ensure that *METIS* produces load-balanced partitions with nearly equal number of states per partition. For all of our benchmarks (in Table 4.1) *METIS* consistently produces connected component partitions that have less than 16 state-transitions between them. The maximum number of outgoing state-transitions from an array determines the radix of the global-switch to be supported.

4.3.3 Case Study: Entity Resolution

Figure 4.6 presents the application of our mapping algorithm to the *space-optimized version* of the *Entity Resolution* benchmark. Entity Resolution is widely used for approximate string matching in databases. The benchmark has 5672 states with 5 connected components (CCs). The largest connected component CC_4 has size 4568 and the smallest one CC_0 has 75 states. Each of the arrays (Array_H and Array_L) in a 16 kB subarray of the LLC slice shown supports 256 states. Our mapping algorithm proceeds as follows. For each unallocated array, starting from the smallest connected component, greedily pack as many connected components in the array (as much as the array can accommodate). If the connected component size exceeds the size of an array, then we invoke the *k-way partitioning* algorithm in METIS for different values of k based on the connected component size.

From the final mapping obtained, it can be seen that a fairly dense packing of CCs is achieved. It can be seen that both CC_0 and CC_1 are allocated the same array. CC_2 takes up a separate array while CC_4 spans across 3 ways. Local switches at each array and global switches for both 1 and 4 ways support intra-array and inter-array state transitions respectively. Furthermore, the densely connected arrays for CC_4 (having many outgoing transitions) are also allocated to arrays in the same way.

4.4 Evaluation Methodology

NFA workloads: We evaluated the proposed approach and mapping schemes on a wide range of benchmark FSMs from the *ANMLZoo* [166] and the *Regex* [43] benchmark suites similar to PAP (Chapter 3). Table 4.1 summarizes some of the important characteristics of these FSMs and the parameters used in our simulations. We used the 10 MB input traces for our evaluation. Similar trends in results are observed for larger inputs.

Experimental Setup: We utilize the open-source virtual automata simulator *VASim* [166] to simulate the proposed architecture. *VASim* allows for fast NFA emulation by traversing paths only for active states. The simulator takes as input the NFA partitions produced by METIS and simulates each input cycle by cycle. After processing the input stream, we use the per-cycle statistics on number of active states in each array to derive energy statistics.

Table 4.2 provides the various delay and energy parameters assumed in our design. To estimate

		Performance optimized				Space optimized			
#	Benchmark	States	Connected Components	Largest CC Size	Avg.Active States	States	Connected Components	Largest CC Size	Avg.Active States
1	Dotstar03	12144	299	92	3.78	11124	56	1639	0.84
2	Dotstar06	12640	298	104	37.55	11598	54	1595	3.40
3	Dotstar09	12431	297	104	38.07	11229	59	1509	4.39
4	Ranges05	12439	299	94	6.00	11596	63	1197	1.53
5	Ranges1	12464	297	96	6,43	11418	57	1820	1.46
6	ExactMath	12439	297	87	5.99	11270	53	998	1.42
7	Bro217	2312	187	84	3.40	1893	59	245	1.89
8	TCP	19704	715	391	12.94	13819	47	3898	2.21
9	Snort	69029	2585	222	431.43	34480	73	10513	29.59
10	Brill	42568	1962	67	1662.76	26364	1	26364	14.29
11	ClamAV	49538	515	542	82.84	42543	41	11965	4.30
12	Dotstar	96438	2837	95	45.05	38951	90	2977	3.25
13	EntityResolution	95136	1000	96	1192.84	5672	5	4568	7.88
14	Levenshtein	2784	24	116	114.21	2784	1	2605	114.21
15	Hamming	11346	93	122	285.1	11254	69	11254	240.09
16	Fermi	40783	2399	17	4715.96	39032	648	39038	4715.96
17	SPM	100500	5025	20	6964.47	18126	1	18126	1432.55
18	RandomForest	33220	1661	20	398.24	33220	1	33220	398.24
19	PowerEN	14109	1000	48	61.02	12194	62	357	30.02
20	Protomata	42011	2340	123	1578.51	38243	513	3745	594.68

Table 4.1: Regex and ANMLZoo benchmarks – with and without common prefix merging.
Number of connected components, states and average active states for Regex and ANMLZoo benchmarks.

	L_switch [L]				G_switch(1 way) [G1]				G_switch(4 ways) [G2]			
Design	Size	Delay	Energy	Area	Size	Delay	Energy	Area	Size	Delay	Energy	Area
CA_P	280x256	163.5ps	0.191pJ/bit	0.033mm^2	128x128	128ps	0.16pJ/bit	0.011mm^2	-	-	-	-
	Number of switches				Number of switches				Number of switches			
	64				8				-			
CA_S	280x256	163.5ps	0.191pJ/bit	0.033mm^2	256x256	163ps	0.19pJ/bit	0.032mm^2	512x512	327ps	0.381pJ/bit	0.1293mm^2
	Number of switches				Number of switches				Number of switches			
	128				8				1			

Table 4.2: Switch parameters for memory-based state transition interconnect

the area, power and delay of the memory array we use a standard foundry memory compiler for the 28nm technology node. The nominal voltage for this technology is 0.9 V. Our 8T crossbar switches are similar to an 8T SRAM array, except without the associated decoding and control logic overheads present in a regular 8T SRAM array. The energy for access to 6T 256×256 cache sub-arrays was estimated to be $22pJ$. The global wire delays were determined using wire models from the design kit using SPICE modeling. Our analysis takes into account cross-coupling capacitance of neighboring wires and metal layers. The global wires have pitch 1μm and are routed on 4*X* metal layers with double track assignment and repeaters spaced $1mm$ apart. The wire delay was found to be $66ps/mm$ and wire energy was found to be $0.07pJ/mm/bit$.

4.5 Results

In this section we first present the speedups obtained by the proposed Cache Automaton (CA) architecture, followed by an analysis of the cache space utilization, energy consumption and reachability of the proposed automaton architecture. As discussed in Section 4.3.1, we evaluate two designs for Cache Automaton. The first design is optimized for performance, and provides lower connectivity. We refer to the performance optimized design as CA_P and the space optimized design as CA_S throughout the results section.

4.5.1 Overall Performance

The overall performance of the *Cache Automaton* is dictated by the clock-period of pipeline. Table 4.3 shows the delay of various pipeline stages across both performance optimized and space optimized designs.

For the performance optimized design (CA_P), the state-match stage accesses 256 STEs. In our proposed architecture modelled after Xeon's LLC slice, each 16 KB data sub-array is 8-way multiplexed. Internally the sub-array is organized into two 8 KB chunks which can operate independently. Each chunk has two halves: Array_H and Array_L which share I/O and 32 sense-amplifiers. Each half consists of 256×128 6T SRAM arrays. A column multiplexer in each half feeds 32 sense-amplifiers, allowing only 32 bits to be read in a cycle per chunk. Thus, together across the two chunks, it is possible to match 64 STEs in a cycle. The SRAM arrays can operate from 1.2 GHz to 4.6 GHz frequency range [90, 55]. We limit the highest possible operating frequency for each SRAM array to 4 GHz or 256 ps cycle time. Thus, four cycles or 1024 ps is necessary to match 256 STEs without sense-amplifier cycling. With our proposed sense-amplifier cycling optimization, the state match time for 256 STEs is 438 ps as shown in Table 4.3.

Design	State-Match	G-Switch	L-Switch	Freq. Max	Freq. Operated
CA_P	438 ps	227 ps	263 ps	2.3 GHz	**2 GHz**
CA_S	687 ps	468 ps	304 ps	1.4 GHz	**1.2 GHz**

Table 4.3: Pipeline stage delays and operating frequency.

The G-Switch stage requires 227 ps composed of 99 ps due to wire-delay and 128 ps due to the global switch. The distance between the SRAM array and global switch is estimated to be 1.5mm assuming a slice dimension of $3.19mm \times 3mm$. The L-Switch stage requires 263 ps. The pipeline clock period or frequency is determined by the slowest stage. Thus the maximum possible frequency for (CA_P) is 2.2 GHz. We choose to operate at 2 GHz. For the space optimized design (CA_S), an operating frequency of 1.4 GHz can be achieved and we choose to operate at 1.2 GHz. The space optimized design is slower due to longer wire delays between the arrays and global switch, and larger global switches.

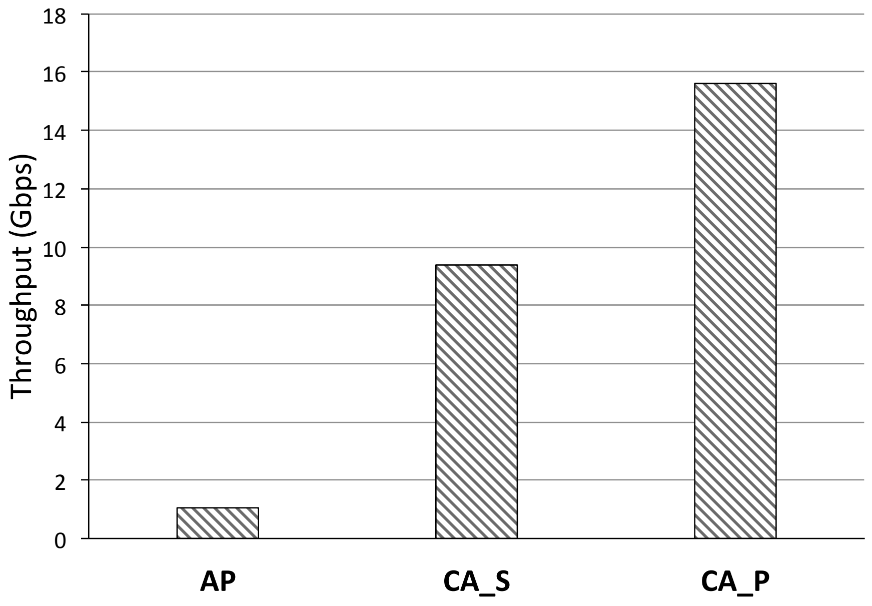


Figure 4.7: Performance - Cache Automaton and Micron's Automata Processor.
Overall performance of Cache Automaton compared to Micron's Automata Processor in Gb/s.

In Cache Automaton, since all state-matches and state-transitions can happen in parallel, the system has a *deterministic throughput of one input symbol per cycle and is independent of the input benchmarks*. This is true for Micron's **Automata Processor (AP)** as well which operates at **133 MHz** frequency. Figure 4.7 shows the overall achieved throughput of Cache Automaton in Gb/s across all benchmarks. Overall, the performance optimized design provides a speedup of $15\times$ over Micron's AP. Prior studies for the same set of benchmarks have shown $256\times$ speedup over conventional x86 CPU [166], thus the Cache Automaton provides a $3840\times$ speedup over processing in CPU. Our space optimized design provides a speedup of $9\times$ over AP.

4.5.2 Cache Utilization

Figure 4.8 shows the cache utilization in MB for different applications considering both the *CA_P* and *CA_S* designs.

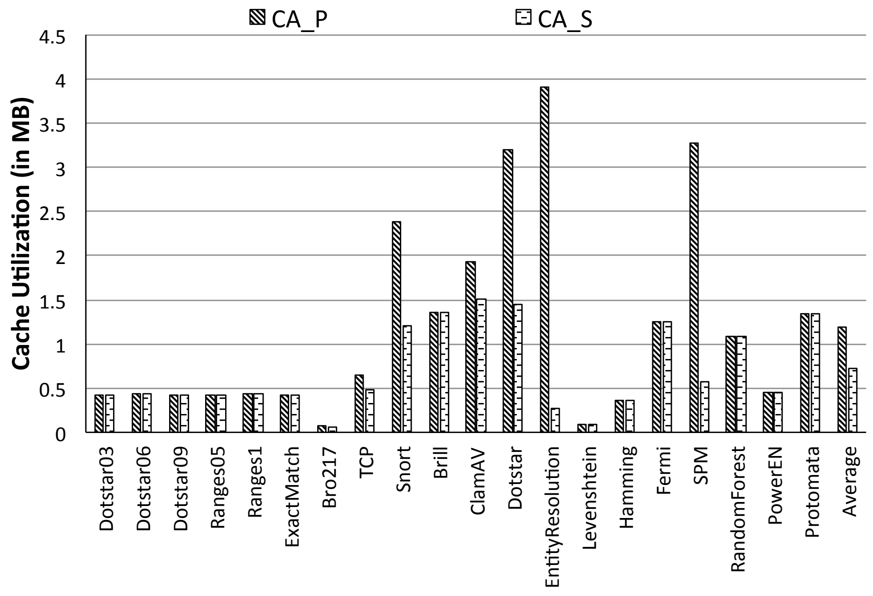


Figure 4.8: Cache utilization of benchmarks.
Cache utilization of benchmarks for the two evaluated designs of Cache Automaton.

The *CA_S* design shows large space savings for Entity Resolution (3.64 MB), SPM (2.7 MB), Dotstar (1.76 MB) and Snort (1.19 MB). The savings achieved compared to *CA_P* are proportional to the redundant state activity in each of the benchmarks. SPM in particular benefits from merging several start states. Although Entity Resolution in the *CA_S* design has only 5672 states, it has high routing complexity with a high average out-degree ($>$ 6 per FSM state). Benchmarks from *Regex* have small connected components and do not show much benefit from *prefix merging*. Random-Forest and Fermi perform a large number of distinct pattern matches and subsequently show lesser state-redundancy with a high number of active states per cycle. These benchmarks show little to no benefit when compared to the *CA_P* design. It must be kept in mind that these space savings can be directly translated to speedup by matching against multiple NFA instances. Also, the *space-*

optimized automata tend to have a lower average active set, on account of lesser redundant state activity, leading to dynamic energy savings. Averaged across all benchmarks, we see that the *CA_P* and *CA_S* designs utilize 1.2 MB and 0.725 MB cache space respectively.

4.5.3 Energy Consumption

This section discusses the energy consumption and power consumption of Cache Automaton. The energy consumption of Cache Automaton depends on two factors. *First*, number of active partitions. Note that even if one STE is active in a partition, it results in an array access and local switch access. *Second*, number of dynamic transitions between partitions, because these result in global switch accesses and expend wire energy. Both these factors are controlled by the mapping algorithms used by the compiler. By grouping STEs based on connected components, the number of active partitions are drastically reduced. By adopting graph partitioning techniques, our compiler successfully reduces the number of transitions between partitions.

Since there is no publicly available data regarding AP's energy consumption, we use an *Ideal AP* energy model which assumes *zero energy for interconnects, routing matrix, and an optimistic 1 pJ/bit for DRAM array access energy*. Conventional DRAMs have been reported to consume anywhere between 2.5pJ/bit to 10 pJ/bit for array access energy (activation energy) [137, 53].

Figure 4.9 shows the energy expended per input symbol for the performance optimized (CA_P) and space optimized (CA_S) Cache Automaton designs, as well as for *Ideal AP* with the same mappings as used by Cache Automaton. Several observations can be made. Benchmarks with higher active state set (see Table 4.1) such as Entity Resolution, SPM, Fermi consume higher energy. These benchmarks also utilize global switches more frequently than other benchmarks. The CA_S mapping has consistently lower energy consumption than CA_P both for Cache Automaton and AP. This is because the CA_S mapping merges many redundant states and thus wastes lesser energy per input symbol on redundant transitions. On average, Cache Automaton (CA_P, CA_S) consumes $3\times$ lesser energy than *Ideal* AP with same mapping (Ideal AP w/ CA_S). The lowest energy consumption of 2.3 nJ/symbol is obtained by the CA_S design, which is $3.1\times$ better

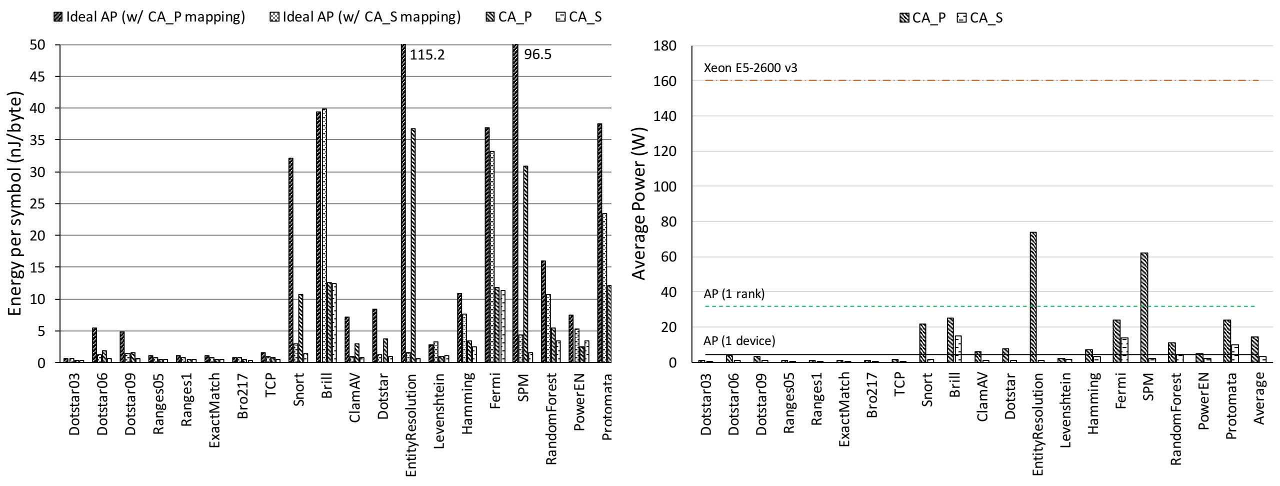


Figure 4.9: Energy and power Consumption of Cache Automaton.
(a) Overall energy consumption of Cache Automaton compared to *Ideal* Automata Processor. (b) Overall power consumption of the two evaluated designs of Cache Automaton.

than best configuration for AP assuming an ideal energy model. Thus, for systems which are energy constrained, we recommend a space optimized mapping. Similar to the Micron AP [15], we also employ partition disabling circuits triggered when there is no active state within a partition, detected using a simple wired OR of all the bits in the active state vector.

Figure 4.9 shows the average power consumption across benchmarks. The power consumption follows the general trends of energy consumption. As expected the power consumption of Cache Automaton is higher, but much lower than TDP of the processor at 160 W (Xeon E5-2600 v3). Thus, we do not expect Cache Automaton to create any power overdrive or thermal problems. Our prototype Cache Automaton which supports NFA processing only in 8 ways of a cache slice can consume a maximum power of 75 W and has capacity to store 128K STEs. Note, CA_S and CA_P have a maximum power consumption of 14.9W and 71.3W. Thus, a system designer can trade off between performance and power.

4.5.4 Reachability and Area Overheads

Memory centric models such as Micron's AP and the proposed Cache Automaton architecture do all the state transitions for an input symbol in parallel. Thus, an important parameter which

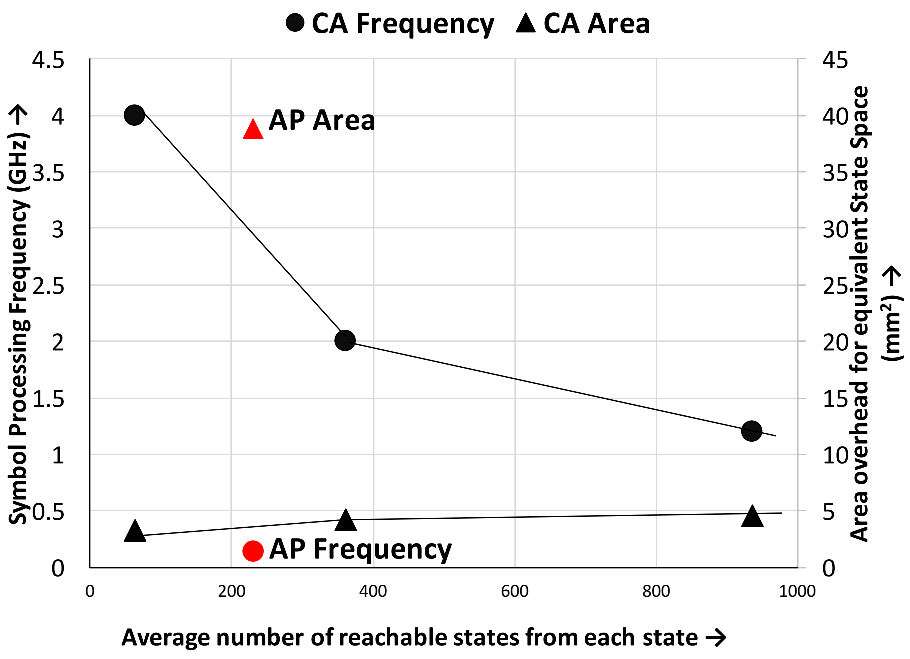


Figure 4.10: Performance, reachability and area tradeoffs for automata processing. Figure shows the performance (frequency for symbol processing), reachability, and area overheads of various Cache Automaton (CA) designs and DRAM-based Automaton Processor (AP).

determines the performance of these architectures and is representative of their scalability to accommodate complex NFAs is the average reachability of a state. Micron's AP provides an average reachability of 230.5 states from any state (Fan-out), while operating at 133 MHz. Figure 4.10 plots the frequency of Cache Automata (left Y-axis) with respect to reachability. A highly performance optimized design can operate at 4 GHz, but only provides a small reachability of 64 states. Larger degree of reachability requires more and bigger global switches, hence has a performance penalty. The proposed CA_P design can operate at 2 GHz while still providing a reachability of 361 states, which is $1.5\times$ better than AP. The proposed CA_S design can operate at 1.2 GHz while providing reachability of 936 states. Note that Cache Automaton supports a maximum of 256 incoming transitions per state (Fan-in), in contrast to only 16 supported by AP.

Figure 4.10 also plots the area overhead of Cache Automaton (right Y-axis) with respect to reachability. Area overhead is reported for supporting a state space equivalent to one AP die (i.e. 48K STEs). The proposed CA_P and CA_S designs incur a modest area overhead of $4.3mm^2$ and $4.6mm^2$ (less than 2% of die area for Xeon E5 server processor which has area of $354mm^2$), but offer high reachability and high performance. In comparison, AP incurs a high area overhead of $38mm^2$ for supporting transition matrix in DRAM dies.

4.5.5 Discussion

This section discusses the impact of various optimizations such as sense-amplifier cycling and parameters such as wire delays. The Table 4.4 column w/o SA cycling shows the frequency of Cache Automaton without sense-amplifier cycling. It can be noted that the pipeline can still operate up to 1 GHz frequency without this optimization. Another alternative to boost frequency without sense-amplifier cycling is to under utilize the cache space and read fewer column multiplexed bits in each cycle.

The proposed designs use global metal layers for connecting the local and global switches. This is motivated by two factors. First, global metal layers are much faster. Second, global metal layers are used only for on-chip networks and are usually underutilized. It is possible to reuse the wires of the hierarchical bus (H-Bus) or H-Tree interconnects used inside a LLC slice. These interconnects are much slower (300 ps/mm [61]). The Table 4.4 column with H-Bus shows the frequency of Cache Automaton when reusing wires of the H-Bus interconnect within a LLC slice. The operational frequency is still $7.5\times$-$11\times$ better than the AP.

Design	Achieved	w/o SA cycling	with H-Bus
CA_P	2 GHz	1 GHz	1.5 GHz
CA_S	1.2 GHz	500 MHz	1 GHz

Table 4.4: Impact of optimizations and parameters.

4.5.6 Comparison with ASIC implementations

The Unified Automata Processor (UAP) [68] and HARE [76] are two recently proposed accelerators that have demonstrated impressive line rates for automata processing and regular expression matching on a number of network intrusion detection and log processing benchmarks. UAP is noteworthy because of its generality and ability to efficiently support many finite automata models using state transition packing and multi-stream processing at low area and power costs. Similarly, HARE has been able to saturate DRAM bandwidth (256 Gbps) while scanning up to 16 regular expressions.

The major advantage of Cache Automaton is its ability to execute several thousand state transitions in parallel (e.g., 128K state transitions in a single cycle using 8 ways of the LLC slice). This massive parallelism enables matching against several thousand patterns (e.g., 5700 in Snort ruleset), while achieving ideal line rate, i.e., 1 symbol/cycle. In contrast, HARE incurs high area and power costs (80 mm^2,125 W) when scanning for more than 16 patterns and UAP's line rate drops for large NFA patterns with many concurrent activations, 0.27–0.75 symbols/cycle [68].

Metric	HARE (W=32)	UAP	CA_P	CA_S
Throughput (Gbps)	3.9	5.3	15.6	9.4
Runtime (ms)	20.48	15.83	5.24	8.74
Power (W)	125	0.507	7.72	1.08
Energy (nJ/byte)	256	0.802	4.04	0.94
Area (mm^2)	80	5.67	4.3	4.6

Table 4.5: Comparison with related ASIC designs.

For fair quantitative comparison, we use Dotstar0.9, containing 1000 regular expressions and $\sim$38K states as used in UAP/HARE for a 10MB input stream. It must be noted that CA can support $>$3000 such regular expressions using less than 8 ways of the LLC slice and shows greater benefits for larger number of patterns and if more ways are used to store NFA. From Table 4.5, it can be seen that CA_P and CA_S provide 3.9$\times$ and 2.34$\times$ speedup over HARE and 3$\times$ and 1.8$\times$ speedup over UAP respectively. While UAP is more energy efficient than CA_P due to efficient compression of state-transitions, CA_S can provide comparable energy efficiency while repurposing the LLC. Note that the 16 kB local memory in UAP can accommodate only few Dotstar0.9 patterns without memory sharing across lanes and we expect several additional DRAM accesses for reading new patterns. This energy is not accounted in Table 4.5. UAP incurs lesser area overhead than CA_P and CA_S, but its 8-entry combining queue may be insufficient to support benchmarks with several thousand active states (e.g., Fermi–4715).

4.6 Summary

This chapter discusses the *Cache Automaton* architecture to accelerate NFA processing in the last-level cache. Efficient NFA processing requires both highly parallel state-matches as well as an interconnect architecture that supports low-latency transitions and rich connectivity between states. To optimize for state-matches we propose a sense-amplifier cycling scheme that exploits spatial locality in state-matches. To enable efficient state transitions, we adopt a hierarchical topology of highly compact 8T-based local and global switches. We also develop a *Cache Automaton* compiler that automates the process of mapping NFA states to SRAM arrays. The two proposed designs are fully pipelined, utilize on an average 1 MB of cache space across benchmarks and can provide a speedup of $12\times$ over AP.

Besides finite-state automata, *Cache Automaton* can also be extended to efficiently process deterministic pushdown automata (or DPDAs). DPDAs provide a rich computation model by extending finite state automata with a stack. They are widely used in parsing different programming languages, serialization formats (e.g., XML and JSON) and mining frequent subtrees within a dataset. Much of DPDA processing can be performed using SRAM array lookups in the last-level cache without involving the CPU. In our recent work, ASPEN [39], we show how the *Cache Automaton* architecture can be customized for DPDA processing. Like *Cache Automaton*, ASPEN uses SRAM array lookups to perform state matches, state transitions and stack updates. ASPEN reduces the latency of XML parsing by $14.1\times$ when compared to optimized CPU XML parser Expat [59] and improves subtree mining performance by $6\times$ when compared to an optimized GPU implementation [150].

CHAPTER 5

Pattern Matching in Genomics

Genome sequencing technology is far outpacing Moore's law in computing. Over the last decade, they have become increasingly cheaper, faster, more portable, and produce longer reads. The cost to sequence a human genome has dropped from \$10 million, a decade ago, to less than \$1000 today. Sequencing providers like Illumina can sequence a human genome for \$600 [24] and BGI/MGI [16] has further reduced the cost to \$100. Apart from the dramatic reductions in cost, there has also been a corresponding increase in sequencing machine throughput. For example, MGI's DNBSEQ-TX and Illumina's Novaseq 6000 produce 20 Terabases [17] and 3.3 Terabases per day respectively [25]. In addition, sequencing no longer requires large bench-top instruments. Oxford Nanopore has introduced the portable MinION sequencer which can produce longer reads (few Kilobases to Megabases) in real-time, although with a higher error rate (5-15%). These portable sequencers also enable a kind of software-defined sequencing paradigm by exposing interfaces to control the length of DNA in real-time as it passes through the pore [109]. Taken together, all these developments have given rise to the widespread usage of genome sequencing and ushered in the era of population genomics with several countries/organizations aiming to sequence the genomes of millions of humans [2, 33, 12]. However, computing solutions, hampered by challenges in scaling transistors, have not been keeping pace. As a result, hardware acceleration approaches and algorithmic developments are essential to keep up with the genomics data deluge.

Genomics is an application domain that extensively makes of pattern matching. In particular, string matching algorithms are commonly used to analyze genomic data. A genome can be considered to be a long string of DNA base-pairs (`bp`) `A, G, C` and `T` (3 `Giga bp` for a human genome). During primary analysis, a sequencing instrument splits the DNA strand into *billions* of short ($\sim$100 bp) strings called *reads*. Secondary analysis aligns these reads to a reference genome and determines genetic variants in the analyzed genome compared to the reference. The alignment step, commonly referred to as *read alignment* in literature, is one of the key computational bottlenecks in secondary analysis [30]. Naïvely matching each read to every location in the reference genome is computationally prohibitive. So, read alignment tools commonly employ the *seed-and-extend* heuristic. This heuristic makes use of two string matching steps: (1) *seeding*: an exact string matching step, that finds a set of promising locations in the reference genome to align the read. It does this by identifying short exact matches called *seeds* between the read and the reference genome and (2) *seed-extension*: an approximate string matching step, that scores each of the candidate locations from seeding to determine the best location to align the read.

Exact and approximate string matching together account for 70-80% of time in read alignment. Building upon the pattern matching acceleration approaches discussed in previous chapters, this chapter presents two hardware-software co-design approaches to accelerate exact string matching (seeding) and approximate string matching (seed-extension) in read alignment.

5.1 Exact String Matching Acceleration (Seeding)

Seeding identifies the set of candidate locations in the reference genome where a possible alignment could exist for a given read. It greatly reduces the computation required during seed-extension, and is important for end-to-end read alignment performance.

5.1.1 Exact Matching Using the FMD-Index

BWA-MEM and BWA-MEM2 find variable-length exact matching seeds between the read and the reference genome. In particular, longest exact matching seeds are found. The longer the seed, fewer the number of locations it occurs in the reference genome. This greatly reduces the work done during seed-extension. Longest exact match seeding works very well for short, high quality reads such as those from Illumina sequencers.

Variable-length exact matching is commonly performed using a highly compressed data structure called the FM-index [117, 71]. The FM-Index consists of: (1) the *suffix array (SA)*, which contains the locations of lexicographically sorted suffixes of the reference genome `R`, (2) the *Burrows Wheeler Transform (BWT)*, computed as the last column of the sorted suffix array of the reference, (3) the *count table (C)* which stores the number of characters in `R` lexicographically smaller than a given character `c` and (4) the *occurrence table (Occ)* which stores the number of occurrences of a character up to a certain index in the suffix array.

Using the count and occurrence tables, it is possible to identify intervals in the suffix array where a particular string exists by performing iterative lookups in these data structures for each successive character. Given the start and end positions of the suffix interval (`s and e`) in iteration `i`, the subsequent start and end positions for iteration `i+1` can be computed using dynamic programming, where, $s_{i+1} = C(c) + Occ(s_i - 1, c)$ and $e_{i+1} = C(c) + Occ(e_i, c) - 1$. Figure 5.1 shows an example of how the C and Occ tables can be used to perform search for query `Q` in reference `R`.

To find exact matching seeds in both strands of DNA, BWA-MEM uses a variant of the FM-Index called FMD-Index, which is built using both strands of DNA.

5.1.2 Seeding Algorithm and Super-Maximal Exact Matches

The seeding algorithm in BWA-MEM2 is based on identifying substrings that have super-maximal exact matches (SMEMs) with the reference genome [118] (Figure 5.1 (a)). A maximal exact match (MEM) is an exact match that cannot be extended in either direction in the read. An SMEM is a maximal length match (MEM) that is not *fully contained* in any other MEM. Short matches lead to an excessive number of hits to be verified by seed-extension, while longer matches can lead to incorrect alignments. BWA-MEM only reports SMEMs greater than a certain minimum length (e.g., 19), empirically determined to be a good trade-off between performance and accuracy.

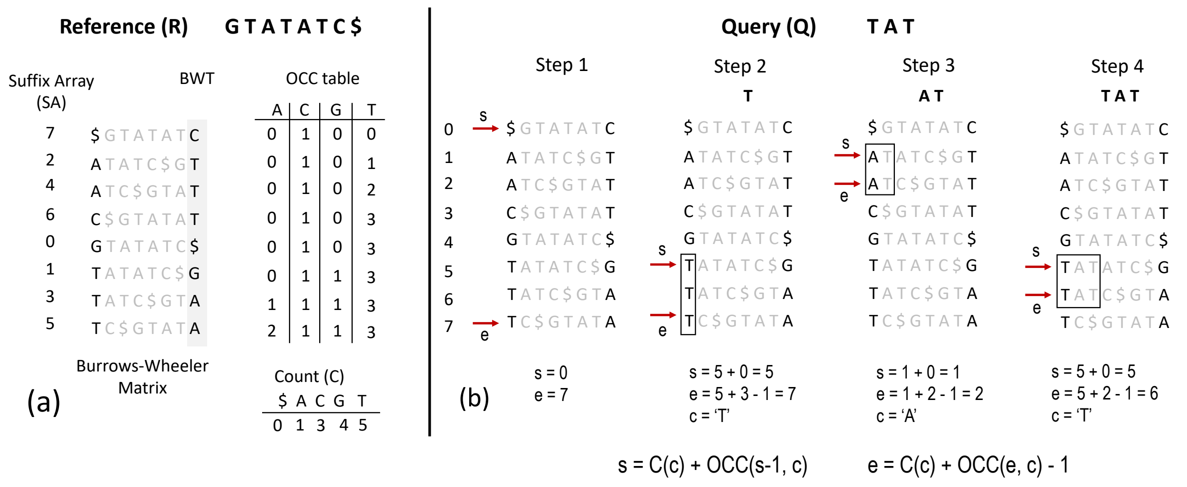


Figure 5.1: FM-Index example
(a) FM-index showing suffix array, occurrence and count tables and (b) Backward search using the FM-index. Note that '$' is assumed to be the lexicographically smallest character.

Figure 5.2 (b) shows the steps involved in determining SMEMs for a sample read and reference pair.

SMEMs are identified in two steps:

(1) ***Forward search:*** For a given query position in the read (e.g., pivot x_0 in Figure 5.2), subsequent base pairs to its right are looked up one at a time in a reference index (e.g., FMD-index) to find the longest exact match in the forward direction. The end position of the longest match becomes the next pivot. During this step, all the positions in the read where there is a *change* in the set of candidate reference locations (hits) are marked (left extension points (LEP) in Figure 5.2 (b). for substrings `T`, `TC`, `TCA` and `TCAGTC`). Only these positions are used as the starting query positions to identify MEMs that extend in the backward direction. Other positions are guaranteed to produce MEMs that are contained within those identified from LEP.

(2) ***Backward search:*** For each query position identified in the previous forward search step as part of LEP (substrings `T`, `TC`, `TCA` and `TCAGTC`), subsequent base pairs to its left are looked up one at a time to find the longest exact match in the backward direction. After this process, SMEMs are identified by discarding MEMs fully contained in other longer matches. In Figure 5.2 (b), `CAATCTCA` and `ATCTCAGTC` are reported as SMEMs. The MEMs `CAATCT` and `CAATCTC` are discarded because they are fully contained in another MEM `CAATCTCA`. SMEMs obtained during

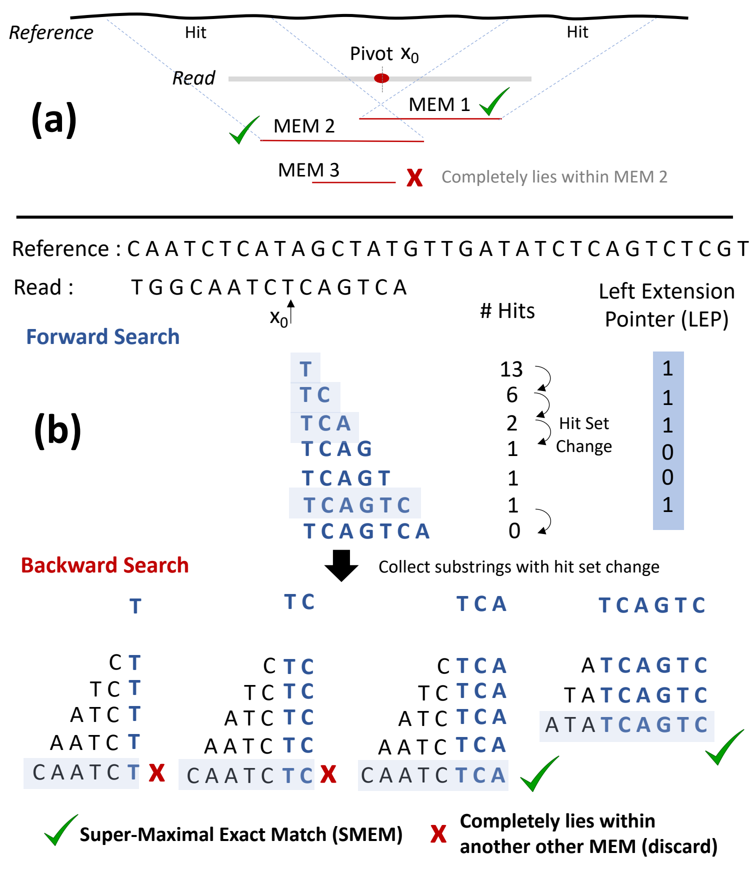

Reference
Hit
Hit
Pivot x_0
Read
(a)
MEM 1 ✓
MEM 2 ✓
MEM 3 ✗ Completely lies within MEM 2
Reference : C A A T C T C A T A G C T A T G T T G A T A T C T C A G T C T C G T
Read : T G G C A A T C T C A G T C A
x_0
# Hits
Left Extension Pointer (LEP)
Forward Search
(b)
T 13 1
TC 6 1
TCA 2 1
TCAG 1 0
TCAGT 1 0
TCAGTC 1 1
TCAGTCA 0
Hit Set Change
Backward Search
Collect substrings with hit set change
T TC TCA TCAGTC
CT CTC CTCA ATCAGTC
TCT TCTC TCTCA TATCAGTC
ATCT ATCTC ATCTCA ATATCAGTC ✓
AATCT AATCTC AATCTCA
CAATCT ✗ CAATCTC ✗ CAATCTCA ✓
✓ Super-Maximal Exact Match (SMEM)
✗ Completely lies within another other MEM (discard)

Figure 5.2: Forward and backward search to identify super-maximal exact matches. (a) Super-Maximal Exact Matches example. (b) Forward and backward search to identify super-maximal exact matches (SMEMs).

seeding are assumed to be part of the final alignment.

BWA-MEM2 also uses two other seeding heuristics to produce highly accurate seeds. The first heuristic known as *reseeding* breaks down long SMEMs ($> 28bp$) that have very few hits (< 10) in the reference genome into shorter substrings with greater number of hits. The second heuristic based on the *LAST* aligner [98] further identifies disjoint seeds in the read using forward search. Use of disjoint seeds reduces the probability that a read is mismapped due to sequencing errors.

5.1.3 Memory-Bandwidth Limitations of Seeding

To identify SMEMs and their locations in the reference genome, both BWA-MEM and BWA-MEM2 use a compressed data structure called the FMD-index [71, 117] which is built using both strands of DNA ($\sim$6 billion characters of the human genome) as discussed earlier. The FMD-index allows the lookup of query `Q` of length `N` in reference `R` using approximately $\mathcal{O}(N)$ memory operations. The FMD-index is utilized for all the three steps of seeding described earlier (SMEM generation, reseeding, and LAST). Detailed descriptions of the FMD-index can be found in [71, 117].

The BWA-MEM2 implementation of FMD-index uses 10 GB (6 GB occurrence table (8$\times$ compressed) + 4 GB suffix array (8$\times$ compressed)) [128] compared to 4.3 GB in BWA-MEM. Further decompressing the occurrence table and suffix array of BWA-MEM2 is shown to only improve performance slightly: 3–7% [128].

Starting from a single character in the read, the FMD-index enables forward and backward MEM searches to determine the number of hits of progressively longer substrings using at most 2 extra memory lookups per character. However, these memory lookups touch different parts of a 10 GB data structure and rarely exhibit spatial locality. This reduces the effectiveness of caching in modern processors and leads to high memory bandwidth requirements. Software implementations of the FMD-index (e.g., BWA-MEM) have attempted to improve the locality of MEM search in two ways. *First*, occurrence table entries are typically co-located with portions of the BWT in tightly packed cache-line aligned data structures to improve the spatial locality of an index lookup. *Second*, backward search passes for substrings sharing the same prefix (e.g., `TC` and `TCA` in Figure 5.2 (b)) are performed in lock-step leading to access of occurrence table data belonging to the same or nearby cache lines [118, 177]. Despite these optimizations, our experiments on real whole human genome reads (details in Section 5.1.6.B.6) show that FMD-index based seeding still has high data requirements (i.e., each read can require $\sim$68.5 KB of index data for seeding). Further, $\sim$40% cycles are spent in core stalling for memory/cache.

FMD-Index based seeding also inherently involves sequential dependent memory accesses and its performance is limited by memory access latency. We mitigate this problem using hardware multiplexing, where one physical compute unit context switches between different reads on a memory stall.

5.1.4 Set Intersection Approach for Seeding

Prior hardware accelerators for seeding directly implement BWA-MEM's FMD-index (Ferragina-Manzini) [117, 71] based seeding, which suffers from poor locality due to irregular memory accesses. We use an implementation that is guaranteed to find all hits as BWA-MEM, but has better locality. Our approach uses an index table that has one entry for each k-mer, which points to a list in a position table [85]. The list contains the *hits* where the k-mer occurs in the reference genome. For each position (*pivot*) in the read, we find a right maximal exact match (RMEM) that is of size at least `k`. To compute RMEM, we determine the hits for the first k-mer starting from the pivot (`H1`). Then we stride by `k`, and find the hits for a k-mer starting at `pivot+k` position in the read (`H2`). We *normalize* these hits to the pivot position by subtracting `k` from their hit values. The set of hits (`H1` and `H2`) are intersected to produce the set of *candidate hits* where we can find the larger string of size `2k`. We can continue this process until the intersection returns an empty set of candidate hits. Then, we reduce the stride progressively from `k/2, k/4, k/8 ..., 1` to compute the RMEM with non-zero candidate hits. We repeat the whole process for each position in the read. The RMEM for the first position in the read is an SMEM. If an RMEM for a later position is a substring of a previously discovered SMEM, it is not reported as a seed, as it is not an SMEM. Seeding returns the hits of all SMEM seeds to the seed-extension step.

5.1.5 Enumerated Radix Trees for Seeding

In this section, we describe an alternative memory-bandwidth friendly approach to seeding that trades off memory footprint for reducing memory bandwidth.

5.1.5.A K-mer Enumerated Index

FMD-index stores a compressed representation of the set of *all suffixes that exist in the reference genome* in lexicographic order. We now consider a substring of length k in the read (referred to as a *k-mer*). Due to natural genome variation and sequencing errors, not all k-mers will exist in the

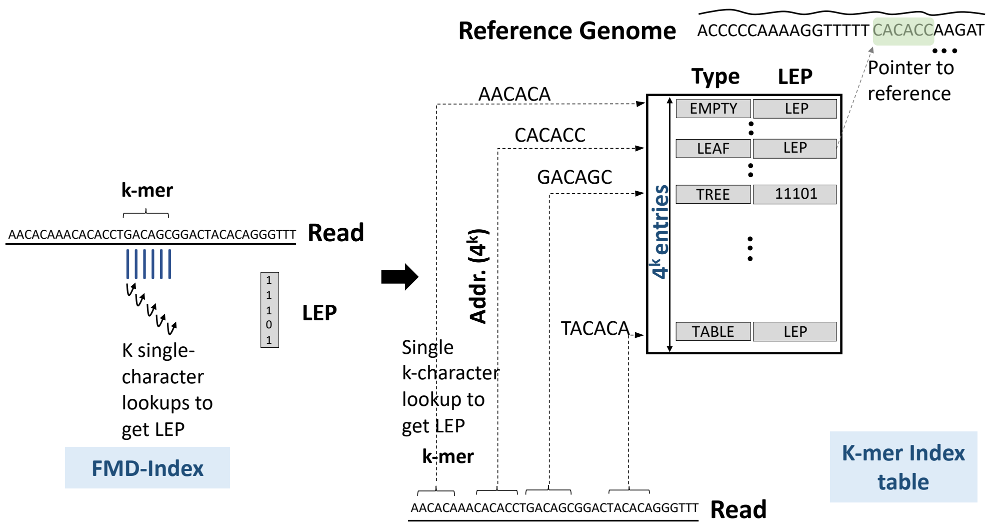


Figure 5.3: K-mer index table compared to FMD-index.
The K-mer enumerated index table pre-computes LEP for each K-mer and reduces K single-character DRAM lookups in the FMD-index to a single K-character DRAM lookup.

reference and, hence, in the FMD-index. Therefore, when looking up a k-mer in the FMD-index, we must start with a 1-mer and grow the string, character by character, for as long as it exists in the FMD-index, or till we reach the desired k-mer length. This iterative, character-by-character access to the FMD-index substantially increases the required number of DRAM accesses, creating a memory bottleneck. This is further aggravated by the fact that accesses to the index rarely follow lexicographic order, making it difficult to exploit locality over such a large window (i.e., set of all suffixes of the k-mer).

To overcome these two limitations, we instead enumerate *all possible k-mers* (whether they exist in the reference or not) and store them in an index table. For each k-mer (an index entry), we then store all its suffixes in the reference. Since all possible k-mers are represented in the index, *k characters* from the read can be looked up in a single memory access, significantly reducing the number of DRAM accesses. Furthermore, subsequent accesses to the suffixes of the k-mer have much improved spatial locality, since they are co-located together. LEP information for the k-mer, resulting from each of the *k* single character lookups is pre-computed and stored in the index table entry. Figure 5.3 shows an example index table enumerating all 6-character substrings. To **choose k**, we observe that BWA-MEM/BWA-MEM2 only report SMEMs greater than a certain minimum length (e.g., 19). This is because shorter substrings lead to an excessive number of hits to be verified by seed-extension. Thus, `k` can be set to any value less than 19. The higher we set it, the more characters can be looked up at once, but it would require more space. We choose $k = 15$ to keep the size of index table tractable ($O(4^k)$), i.e, 1 Giga entries when $k = 15$. Later, in Section 5.1.5.C we discuss a solution to effectively increase k by selectively using a multi-level index.

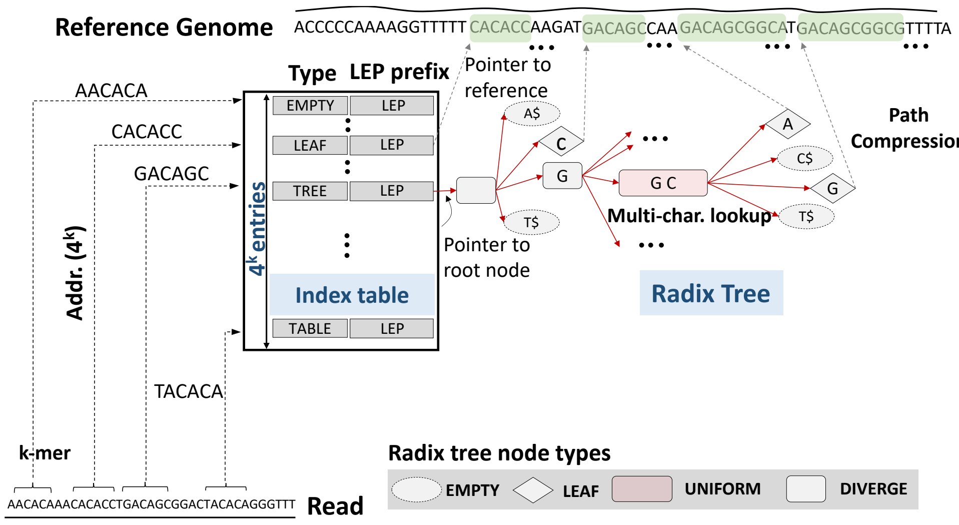


Figure 5.4: Enumerated Radix Tree (ERT) with index table and space-optimized radix tree.

5.1.5.B Customized Radix Tree

The next question is how to store the suffixes of a k-mer in an index entry, so that we can support MEM searches for strings longer than `k`. One option is to augment the index table with an FMD-index, and iteratively grow the k-mer prefix. However, even within the subset of all suffixes sharing the same k-mer prefix, FMD-index lookups have poor locality. Also, they still operate with a single character at a time.

To overcome this problem, we observe that a radix tree can naturally support *multi-character lookups*. This is because in a radix tree, we can merge all singleton paths into a single node, thereby addressing a multiple character lookup with a single memory access. Figure 5.4 shows a radix tree for one k-mer `GACAGC` in the index table (note radix is 4 for the genome alphabet). The proposed

ERT merges singleton paths (`GC` in Figure 5.4) using variable-size internal nodes that store the full singleton path string (designated as `UNIFORM`). A singleton path is encountered when all paths in the tree from a certain node onward share a common string.

Early Path Compression: To further improve the space-efficiency of the ERT, we observe that a k-mer frequently becomes *unique* in the reference genome as it increases in length. This means that, past a certain length, a prefix is followed by a single, unique suffix string in the reference genome. This would introduce a `UNIFORM` node in the ERT with a singleton string of characters (up to the length of the read). To avoid storing this long string, we instead replace it with a pointer to the occurrence of this string in the reference genome. In Figure 5.4 we show how in the ERT, these nodes are marked as *leaf nodes*, containing a single pointer. Leaf nodes encountered during a MEM search are decompressed, by fetching the full reference string corresponding to the reference pointer stored at the leaf node. Note that the pointer in the leaf node is required regardless of this compression technique since it is necessary to indicate the location of the traversed k-mer in the reference genome. Hence, it does not present any storage overhead. Instead, this optimization results in $\sim 2\times$ space savings and was critical for being able to store the full human genome in under 64 GB of storage, which is a common configuration for servers.

5.1.5.C Supporting large K at low space overhead

Enumerating all k-character prefixes in the index table can have prohibitive space overheads for large k. For example, a 19-mer table has 4^{19} entries, resulting in 2 TB of space, assuming 8 bytes per entry. However, the human genome is not a random string of characters from the genome alphabet. The repetitive nature of the human genome makes the distribution of hits (or leaf nodes in the radix tree) for different k-mers heavily skewed.

Our key observation is that the skewed distribution of k-mers in the human genome can be used to design a multi-level index table. For a given number of hits `X`, Figure 5.5 shows the number of k-mers in the human genome that have hits $>$ `X`. It can be seen that very few k-mers ($\sim 0.01\%$) have greater than 1000 hits. However, these k-mers have dense radix trees, which can be compactly

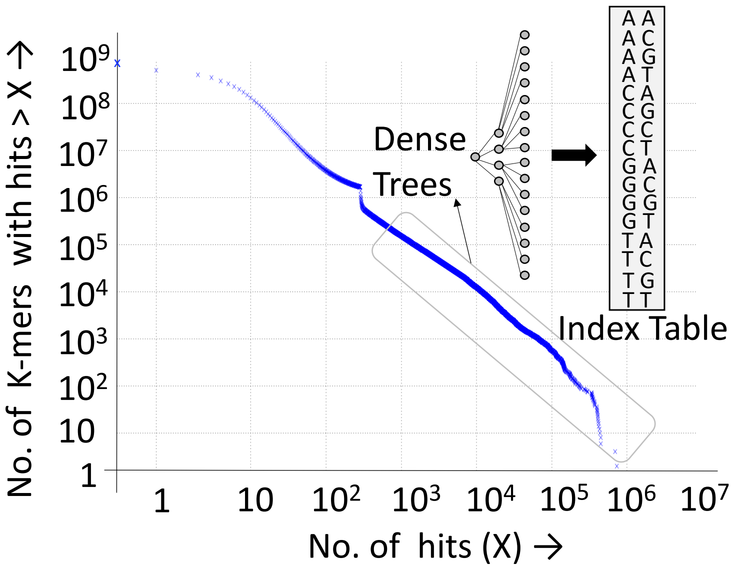


Figure 5.5: Figure showing the skewed hit distribution for k-mers.

represented using an index table as shown in Figure 5.5.

Instead of enumerating all k-character prefixes for large k, we decompose the index table into two levels (Figure 5.6 ⑦), wherein the first level enumerates all k-mers and the subsequent level enumerates all x-character suffixes for a subset of k-mers (such that $k + x$ = min. SMEM length). The multi-level index table further extends the benefit of multi-character lookup. While choosing a larger x helps reduce tree traversal time, for the human genome we were able to accommodate up to $x = 4$ (fan-out = 256) for a subset of 15-mer dense trees without increasing space overheads. Compared to $x = 1$, $x = 4$ improves CPU performance by 10%. Since most trees are shallow (83% of leaf nodes have depths $<= 8$), we did not explore more than two-levels or higher fan-out for the internal nodes of ERT.

5.1.5.D ERT index table entries and radix tree nodes

This section describes details of different entries in the ERT index table as well as the different types of radix tree nodes present in Figure 5.6.

Each index table entry in ERT contains a `Type` field, `k-1 bit LEP` field and a `pointer` field that indicates the address of the root node of a radix tree (not shown). `Type` can be any of the following: (1) EMPTY: k-mer is absent in the reference. (2) LEAF: k-mer is unique (i.e., has the same suffix at all occurrences in the reference). (3) TREE: k-mer exists in the reference and has a pointer to the root of the radix tree. (4) TABLE: k-mer has a large number (> 256) of unique suffix strings in the reference. The index table entry for this k-mer points to a 2^{nd}-level index table to succinctly represent these suffix strings. ERT represents a singleton path using a variable-size internal node (UNIFORM) supporting multi-character lookup (5). If a singleton path ends in a leaf, it is truncated at its start with a LEAF node that points to the reference genome (path compression), 6). Each path from the root to the leaf in ERT encodes a prefix of a sequence occurring in the reference genome. To indicate absence of prefixes in the reference, ERT also includes EMPTY nodes (ending with $). UNIFORM nodes have only one valid child branch for the prefix, while DIVERGE nodes have more than one.

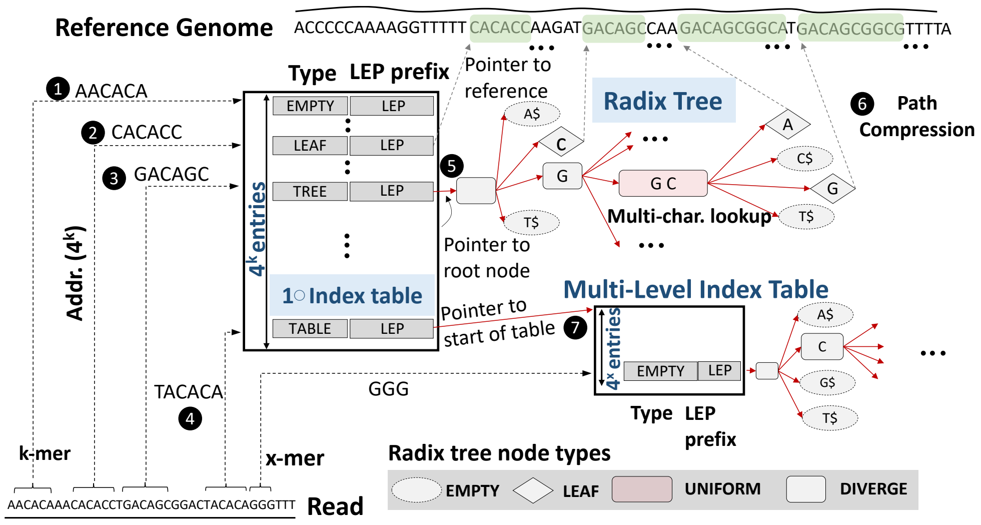


Figure 5.6: Multi-level index table in ERT to support large K at low space overhead.

5.1.5.E ERT Construction

The ERT k-mer index table and corresponding radix trees are built by first enumerating all possible k-mers and then querying a pre-built FMD-index of the reference genome to grow the trees for each k-mer according to all existing sequences in the reference. Each k-mer and ERT path from the root to the leaf of the tree corresponds to a unique sequence in the reference. The locations of these sequences are stored as pointers at the leaves of the tree, as noted above. Note that if a particular k-mer does not exist (referred to as `EMPTY` in Figure 5.6), we do not store a pointer to an ERT tree since no SMEM with length $k < 19$ is required. In our implementation where $k = 15$, 38.8% of the index entries are empty. For an `EMPTY` entry, we still compute the LEP bits corresponding to the k-mer and store it in the index table, to indicate at which positions along this k-mer a backward traversal must be initiated.

The size of the ERT index depends both on the size and the repetitiveness of the reference genome under study. We empirically estimate the space occupied by the ERT index to be $\sim$20 N bytes, where N is the size of the reference genome in Giga base-pairs. For instance, the ERT index size for the human genome is 62.1 GB (index table–8 GB; radix tree–54.1 GB) and for the wheat genome ($\sim$17 Giga bp) is 320 GB. ERT index construction is fast and takes $\sim$1 hr wall-clock time for the human genome with 72-threads. Note that ERT index construction is not a bottleneck, since it is done only once per reference genome and reused across several read alignment runs.

5.1.5.F Finding longest exact matches in forward direction with ERT

Once constructed, we can use the ERT to search for MEMs according to the SMEM seeding algorithm (Section 5.1.2).

For a given `k-mer` scanned from the pivot position in the read, we do the following: **(1)** The index table is looked up using the `k-mer` with a single DRAM access (① in Figure 5.7). If an entry in the index table exists, the root of the `k-mer` tree is also fetched with a second memory access. **(2)** To search for matches longer than k, we traverse the nodes of the `k-mer` tree based on the remaining base pairs in the read (②). We continue traversing the tree either until a leaf node is encountered or an empty node is reached (i.e., there is no branch in tree for the particular character in the read). **(3)** If a leaf node is reached, the reference sequence corresponding to that leaf is fetched with a DRAM access to determine the final characters matching with the read (③). **(4)** If we reach an (`EMPTY` node) in the tree, we have found the maximal exact match (MEM) starting rightwards from the pivot. At this point, all locations where this MEM exists in the reference (i.e.,

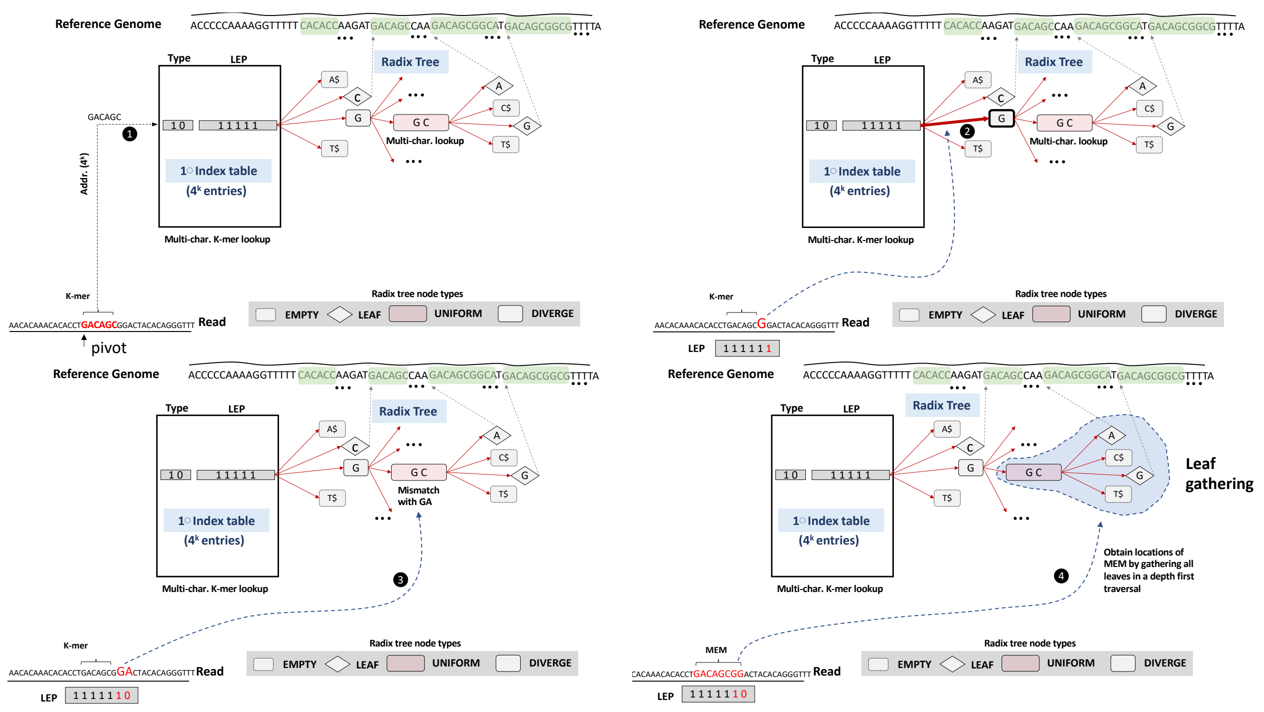


Figure 5.7: Forward search for maximal exact match (MEM) with ERT.

all leaf nodes in the downstream sub-tree) are gathered using a depth-first traversal, referred to as *leaf gathering* (④). Note that while the sub-tree can be traversed in any order to gather all leaves, we chose depth-first traversal to obtain hits in the same order as BWA-MEM and BWA-MEM2. **(5)** Recall from Section 5.1.2, that during forward search using the FMD-index, we keep track of changes in hit sets and obtain the LEP bitvector for the MEM. LEP bits that have been set ("1") are used to initiate backward search from the corresponding positions in the read. So far, from the index table entry we obtain only the LEP bits for the k-mer. We compute the LEP bits for the remaining characters in the MEM as follows. Each time a `DIVERGE` node is encountered during the traversal of an ERT path, a bit is set in the LEP bitvector since this indicates a *hit set* change, i.e., the hits are divided across the divergent paths from that node.

5.1.5.G Supporting Backward Search with ERT

In the above, we have discussed how ERT can support forward search. For exhaustive identification of all the SMEMs in the read, the forward search procedure must be repeated starting from every position in the read. This is wasteful and can lead to redundant computation. However, by supporting backward search in the same index, we can begin seeding only from those read positions at which hit sets changes have been recorded during forward search (stored as LEP).

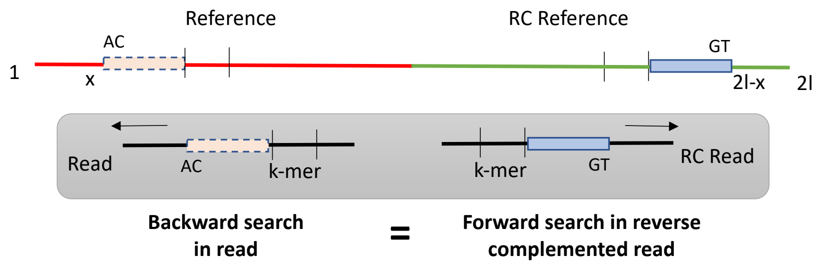


Figure 5.8: Bidirectional search.
Indexing both forward and reverse complemented reference genome to enable bidirectional search. Backward search for pattern can be emulated as a forward search of the corresponding reverse complemented pattern.

To support backward search, we make the observation that the two strands of DNA in the human genome are reverse complements of each other. Since, we are unsure if the read originated from the forward or reverse strand, we index both strands in the same index. This means that as shown in Figure 5.8, backward search for a pattern from the read can be emulated by using forward search of reverse complemented pattern in the reverse complemented read. This is similar in principle to the FMD-index used in BWA-MEM. An alternative strategy is to build two separate indexes, one each for the forward and reverse complemented strands, however, this approach requires two lookups per k-mer. In space constrained scenarios, where only one of the strands can be indexed, SMEMs can be identified at the cost of doubling the number of index lookups. This is because both the forward and reverse complements of the k-mer have to be looked up in the index. Backward search can be supported by doing forward search on the reverse complemented read as before.

5.1.5.H Re-using MEM Searches with Prefix-Merged Radix Trees

The goal of prefix-merged radix trees is to reuse work across MEM searches (forward or backward) from consecutive positions in the read.

In the seeding computation, the time spent doing backward MEM searches is $\sim 2\times$ that of

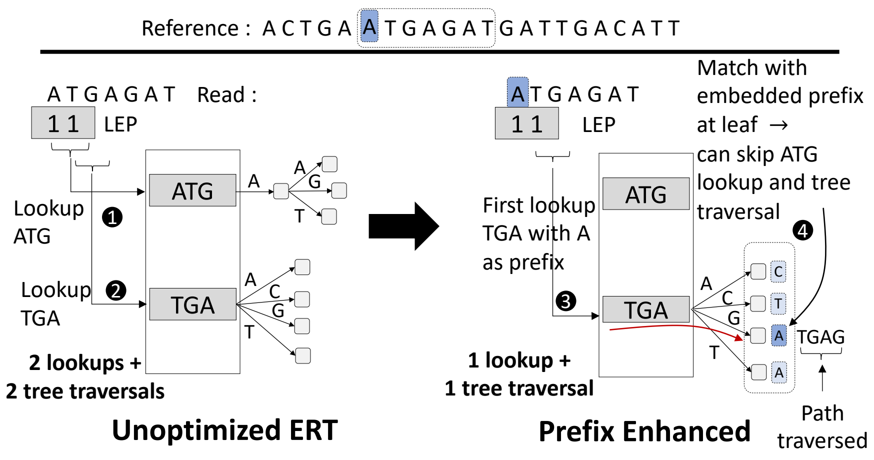

Reference : A C T G A A T G A G A T G A T T G A C A T T
A T G A G A T Read :
1 1 LEP
Lookup ATG
❶
Lookup TGA
❷
ATG
TGA
A G T
A C G T
2 lookups + 2 tree traversals
Unoptimized ERT
A T G A G A T
1 1 LEP
First lookup TGA with A as prefix
❸
Match with embedded prefix at leaf → can skip ATG lookup and tree traversal
❹
C T A A
TGAG
Path traversed
1 lookup + 1 tree traversal
Prefix Enhanced

Figure 5.9: Reusing MEM searches with prefix-merged radix trees.
Merging radix-trees by adding prefix data at the leaf nodes allows ERT to leverage prefix information to perform multiple MEM searches in a single tree traversal.

forward search, making it important to optimize this step. On average, we find that there are ∼10 backward searches for each forward search from a pivot. Also, it is common to observe searches from adjacent query positions in the read (consecutive bits of LEP are ’1’). Normally, these lead to multiple independent index table lookups and tree traversals as shown in Figure 5.9.

In the unoptimized ERT, there exists a radix tree for each k-mer that occurs more than once in the reference, including adjacent, sliding window k-mers (e.g., `ATG` and `TGA`). We recognize that radix trees for adjacent k-mers contain redundant information and that the information contained in one of the trees can be reconstructed from the adjacent k-mer’s tree by storing prefix information at each of its nodes. In the example shown in Figure 5.9, two MEM searches need to be performed starting from the first `ATGAxyz` and second positions `TGAxyz` of the read (see LEP). Each MEM search involves an index table lookup and tree traversal (① and ②), resulting in 2 index table lookups and 2 tree traversals in total.

The unoptimized ERT does not provide an option to reuse work across multiple MEM searches from consecutive positions in the read. However, we find that there does exist opportunities to reuse work across multiple MEM searches, because of overlap in characters from the read that are used to traverse ERT. For example, it is possible to compute the results of traversing the `ATG` tree by traversing the `TGA` tree alone, if while traversing the `TGA` tree, we can simultaneously determine if there are leaves in the `TGA` tree with `A` as a prefix.

We now discuss extensions to the unoptimized ERT to enable checking for prefix matches during traversal. First, it must be noted that if any of the leaf nodes of an internal node's subtree contains the desired prefix character, then the internal node also contains the prefix character. While storing prefix information at internal nodes does have the benefit of terminating some MEM searches early in case of prefix mismatch, augmenting each of the internal nodes in ERT with this prefix information takes up significant space. Therefore, in our prefix optimized ERT, only leaf nodes are augmented with prefix characters (2 bits per prefix character) found at the corresponding reference positions (Figure 5.9). Storing prefix information at the leaf nodes is sufficient as prefix information at each of the internal nodes can be reconstructed by visiting all of the leaf nodes in its corresponding sub-tree.

Our key observation is that with such a prefix-merged radix tree, multiple backward searches (`TGAxyz` *and* `ATGAxyz`*) can be performed in a single index table lookup and tree traversal by checking for prefix character matches at each visited node.* In Figure 5.9, when we reach the leaf node represented by string `TGAG` (③), we can simultaneously match character `A` from the read as a prefix, resulting in the MEM represented as `ATGAG` (④). This reduces the two MEM searches in the unoptimized ERT into one with the prefix enhanced version of ERT.

The example in Figure 5.9 uses a 1-character prefix at the leaf nodes. We chose a 1-character prefix after observing that each backward search on average matches $\sim$1 prefix character at the leaf nodes. With the help of prefix-enhanced radix trees, we were able to reduce the number of backward searches performed by 50%.

5.1.5.I Locality with K-mer Reuse

In Section 5.1.5.H, we saw how work can be reused across multiple MEM searches from consecutive positions in the read. We observe that another opportunity to reuse work across MEM searches comes from the fact that it is common to initiate MEM searches for the same k-mer as part of different reads. For a batch of 1000 reads, we observe that $\sim$45% index table and radix tree

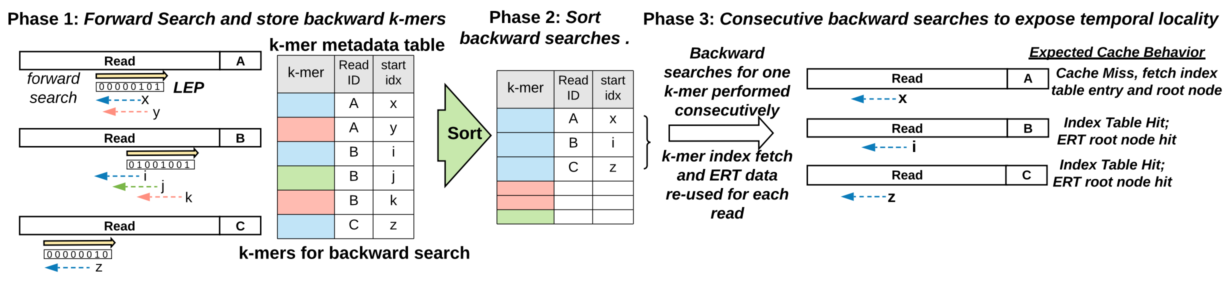

Phase 1: Forward Search and store backward k-mers
k-mer metadata table
Read A
forward search
LEP
x
y
Read B
i
j
k
Read C
z
k-mer
Read ID
start idx
Sort
k-mers for backward search
Phase 2: Sort backward searches .
Phase 3: Consecutive backward searches to expose temporal locality
Backward searches for one k-mer performed consecutively
k-mer index fetch and ERT data re-used for each read
Expected Cache Behavior
Cache Miss, fetch index table entry and root node
Index Table Hit; ERT root node hit
Index Table Hit; ERT root node hit

Figure 5.10: K-mer reuse algorithm for leveraging temporal locality across MEM searches. **Phase 1)** Perform forward extension for a batch of reads. Identify all k-mers required for backward extension (dashed lines) using LEP. **Phase 2)** Sort k-mers to bring backward extension tasks involving the same k-mer together. **Phase 3)** Perform all backward extensions involving the same k-mer together to exploit locality.

accesses from k-mers can be reused, with reuse improving slightly with larger batch sizes. This is expected given the highly redundant nature of the human genome and high coverage of sequenced reads needed to correct sequencing errors (each position in the reference genome can be covered by 30–50 reads on average). However, in the original seeding algorithm using ERT, typically several radix trees need to be accessed to find seeds for a read, and their aggregate size exceeds that of on-chip caches. As a result, a radix tree usually gets evicted before it can be reused, resulting in a low hit rate in traditional caches. This problem can be mitigated if we can look at not a single read alone, but a large batch of thousands of reads and determine in advance the set of k-mers for which we will need to perform radix tree node fetches from DRAM.

Recall that the set of k-mers from which backward MEM searches are initiated in the original seeding algorithm depend on the LEP bitvector computed during forward search. But in the original seeding algorithm, one forward search pass is performed for each pivot position in the read followed by multiple backward search passes. *Our key observation is that the forward and backward search phases of the seeding algorithm need not be performed sequentially for each read. Instead they can be decoupled to expose temporal locality.* More specifically, we can first perform forward search for a batch of thousands of reads and then identify all the unique k-mers that are to be used in backward search (using LEPs). Later, we fetch each radix tree once for each unique k-mer and perform all backward searches for that k-mer before moving to the next k-mer. In this way, all backward MEM searches involving the k-mer across the batch of reads now only involve accesses to the on-chip cache as opposed to DRAM. We refer to this technique that reorders the forward and backward search passes to better expose temporal locality during MEM search as *k-mer reuse*.

Figure 5.10 describes the steps to be performed to leverage k-mer reuse across multiple reads. While processing the forward searches for a batch of N reads, we store each backward search that must be computed in a k-mer metadata table implemented on-chip (Phase 1). Each backward search entry is composed of: **(1)** k-mer starting from the backward search point in the read, **(2)** the read ID in the batch, and **(3)** start position of backward search in the read. Once all forward searches have been completed for a batch of reads, we sort all entries in the metadata table (Phase 2). In this way, we group each required backward search by k-mer. We then proceed one k-mer at a time and compute all backward searches associated with a k-mer sequentially (Phase 3). The first time a k-mer is encountered, we perform one index table lookup, as well as fetch a portion of the k-mer's tree into an on-chip cache. Subsequent backward extensions then consult this cache during tree walking, skipping two otherwise mandatory DRAM accesses. We find that forward, backward and sort phases of the k-mer reuse based seeding algorithm take 26.4%, 67.6% and 6% time respectively.

5.1.5.J Tiled Layout for Spatial Locality

Similar to [99, 57], the spatial locality of ERT accesses can be improved by using a tiled layout for radix tree nodes as shown in Figure 5.11. In this layout, sub-trees of nodes that are likely to be accessed at the same time are clustered together into a single cache block- or a DRAM page-sized tile. Compared to breadth-first or depth-first layout of nodes, the tiled layout guarantees at least $log_4(n + 1)$ nodes accesses per tile, where n is the number of nodes in the tile. With this optimization, ERT traverses $\sim$3 nodes on average per 64 B, utilizing 50% of the data it fetches from memory.

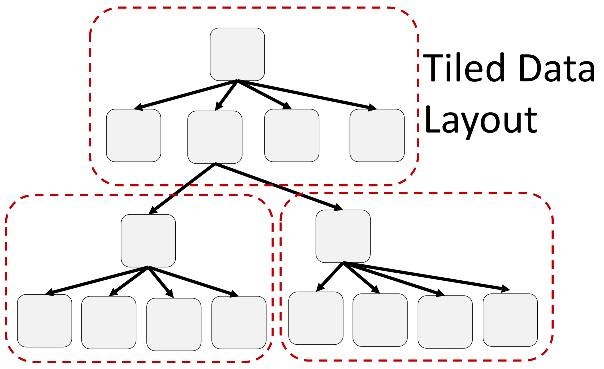


Figure 5.11: Cache-friendly tiled data layout for ERT.

5.1.5.K Pruning Backward Searches: Zigzag Seeding

After performing forward search from the pivot (Figure 5.12 ①), we obtain the LEP vector ② to guide backward search ③. Typically backward search is performed starting from each query position in the read where the set of candidate hits changes (as given by set bits in the LEP vector). In the original seeding algorithm, backward search proceeds in the right-to-left order for each bit set in the LEP vector, starting from the longest match (rightmost '1' bit in LEP) and ending at the shortest match (leftmost '1' bit in LEP). However, as can be seen in Figure 5.12 (a), many of these backward extensions end in MEMs that are fully contained in previously identified SMEMs. These MEMs are discarded by the seeding algorithm.

Performing backward extensions for LEP positions that do not lead to SMEMs is wasteful. Ideally, we would like to perform only those backward extensions that lead to non-overlapping MEMs. To achieve this, we redesign the seeding algorithm to alternate between forward and backward search in a *zigzag* fashion as shown in Figure 5.12 (b). Instead of starting backward search at the rightmost set LEP position, we start backward search at the pivot and extend leftward until no longer match can be found. We later extend the same match beyond the pivot in the forward direction until no longer match can be found. Backward searches from LEP positions in the read that lie within the forward match can safely be skipped since they are guaranteed to produce shorter fully-overlapping matches. The interleaved backward-forward search is repeated from the next set LEP position beyond the forward match as shown in Figure 5.12 (b).

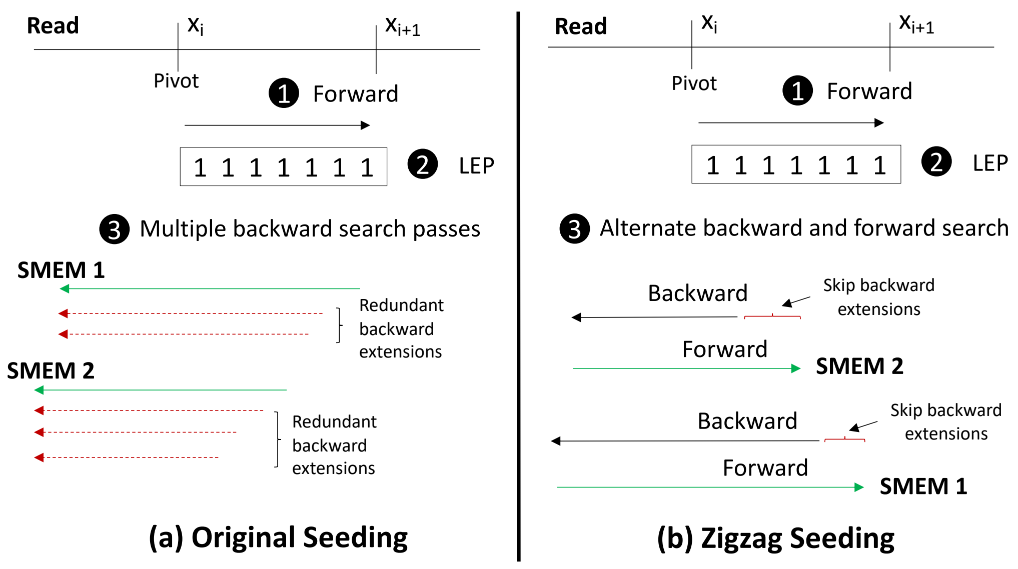
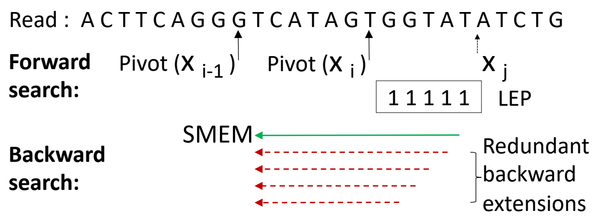


Figure 5.12: Zigzag seeding to reduce backward searches.
(a) Original seeding algorithm that performs multiple backward search passes sequentially leading to redundant backward extensions. (b) Redesigned seeding algorithm that interleaves backward and forward searches to skip redundant backward extensions.

5.1.5.L Pruning Backward Searches: Ordering Backward Searches



Read : A C T T C A G G G T C A T A G T G G T A T A T C T G
Forward search:
Pivot (X_{i-1})
Pivot (X_i)
X_j
1 1 1 1 1
LEP
Backward search:
SMEM
Redundant backward extensions

Figure 5.13: Right-to-left backward search algorithm.
Pruning wasteful backward searches by performing backward searches in the right-to-left order.

Typically backward search is performed starting from each query position where the set of candidate hits changes (as given by the LEPs), in no particular order. However by imposing an order for the backward extension pass, namely starting from the rightmost query position where the hit set changes and proceeding leftward, it is possible to prune out subsequent backward searches as illustrated in Figure 5.13.

The forward pass partitions the read into multiple non-overlapping MEMs. As a result, each backward search is guaranteed to not produce a MEM that spans across multiple pivots. If any backward extension from position x_j in the read reaches the previous pivot x_{i-1}, then backward extensions $\forall x$, where $x < x_j$ are guaranteed to produce MEMs that are contained within that of x_j and are redundant.

5.1.6 Hardware Accelerators for Seeding

In this section, we show how custom hardware can be leveraged to further accelerate both the set intersection and the ERT approaches to seeding.

5.1.6.A Set Intersection Accelerator

We observe that fetching data from the index table and the position table can become a performance bottleneck. To enable greater reuse of the index and position tables across reads during SMEM computation, we segment the genome, and construct index and position tables for each segment. Segmenting also enables the index and position tables to be stored in on-chip SRAM, providing low-latency access, and alleviating the memory bandwidth bottleneck. All the reads are processed for one segment, and then repeated for the next segment.

Intersecting hit sets is a performance-critical operation in determining SMEM seeds and their hits. Our seeding accelerator implements several optimizations to optimize this operation. One, we use 512-entry on-chip CAM per seeding lane to compute intersections. We defined its size based on our empirical analysis of k-mer indices for human genomes that showed that most k-mers have less than 512 hits when `k = 12`. Two, if the set of hits of the current k-mer is larger than 512, we do a binary search. A binary search is possible, because position tables are constructed offline for a reference genome, and therefore we can store the hits for a `k-mer` as a sorted list. Three, we find that a common performance issue is when intersecting the hits of the first two k-mers starting from a pivot. We mitigate this problem as follows. Instead of striding by `k` for the second k-mer, we lookup several k-mers with lower strides. We select the k-mer with the smallest hit set, intersect it with the first k-mer, and continue the RMEM process after that k-mer. Since the size of intersected candidate hits can only decrease, starting RMEM with a small number of hits can

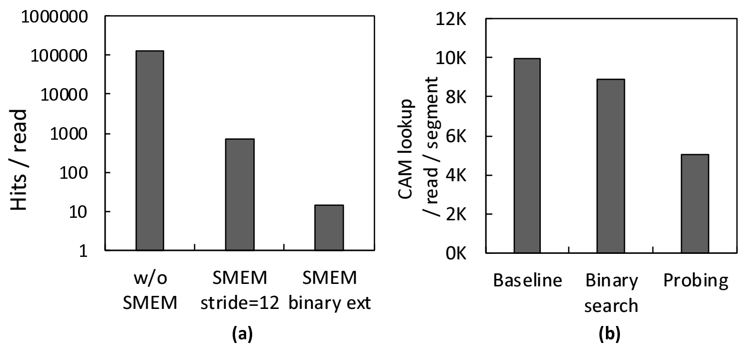


Figure 5.14: GenAx seeding accelerator optimizations.
(a) Reduction in hits per read. (b) Reduction in CAM lookups per read per segment.

reduce the overall number of CAM lookups during the rest of the RMEM computation. Four, we use a variant of the above optimization for quickly seeding reads that have exact matches in the reference genome. We observed that for real world human genome datasets consisting of nearly 1.5 billion short reads, $\sim$75% of the reads have exact matches in the reference. SMEMs for these reads do not need to be verified by seed-extension. To optimize for this common case, for each read, we lookup the index for a set (of size $\lceil readlength/k \rceil$) of k-mers that span the entire read starting from its beginning, where each k-mer is offset by k. We select the smallest hit set, and then start intersecting with the next smallest, and complete the intersection with all the sets. If the intersection of hit sets of all these k-mers result in a non-empty set, then we have found an exact match for the read in the reference, and therefore we can skip the rest of the above steps for it.

Seeding performance breakdown: While we could have used Burrows-Wheeler Transform (BWT), one of the mainstream solutions for genetic string indexing, it suffers from irregular memory accesses. Naive implementations of a hashing solution, on the other hand, require handling a large number of hit positions in return for better locality. Figure 5.14 (a) shows the average number of hits generated by the hash table (lower is better for processing). We observe our proposed optimizations, i.e., SMEM and binary extension, can filter out the insignificant hits, resulting in reduced workload for the SillaX machine downstream by orders of magnitude.

Figure 5.14 (b) presents the reduction of CAM lookups from position table lookup optimizations. Since binary lookup of the position table results in logarithmic search time, the number of

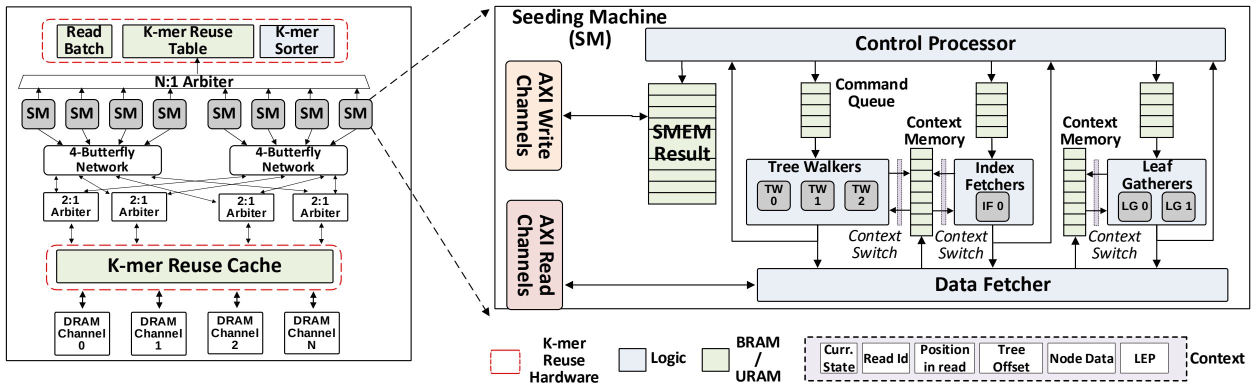


Figure 5.15: ERT seeding accelerator architecture.
Seeding Accelerator architecture. Each Tree Walker (TW) is responsible for scanning a read, walking ERT Trees, and computing candidate SMEMs. Each Tree Walker can switch between multiple contexts to help hide memory latency. The Data Fetcher (DF) is responsible for serving ERT and reference fetch requests to DRAM. The Control Processor (CP) coordinates read fetch, and k-mer reuse phases.

CAM lookups also decreases in proportion to the search time. Moreover, since certain k-mers are known to have large number of hit positions (e.g. AA...A and ATAT...A), probing effectively helps to find a better starting point with a k-mer having fewer hit positions, reducing the overall CAM lookups.

5.1.6.B Enumerated Radix Tree Traversal Accelerator

5.1.6.B.1 Overview The overall architecture of our ERT-based seeding accelerator is shown in Figure 5.15. The accelerator is composed of multiple parallel seeding machines connected to the available DRAM channels using a crossbar network. Each seeding machine is composed of a control processor that issues commands to three types of processing elements. Each processing element (PE) is provisioned with multiple lightweight contexts and performs a sub-task associated with SMEM identification (i.e. index table lookups, walking ERTs, and depth-first search based leaf gathering). When a processing element issues a memory request to the Data Fetcher–a rudimentary address generation unit and memory controller– and a memory stall occurs, the processing element immediately switches to a new context. This fine-grained context switching greatly increases compute density of each seeding machine and is essential to an FPGA implementation with limited logic and routing resources. When the memory request returns, its data is stored in the corresponding PEs context memory and the context is marked as ready.

Decoding radix-tree nodes to determine the next node while traversing the ERT and control operations in the SMEM algorithm are the most time consuming compute steps in ERT-based seeding. Prior to designing custom functional units for these steps, we explored the RISC-based Xilinx MicroBlaze softcore. However, on the MicroBlaze, node decoding resulted in 10–16× higher latency based on ERT node type and required 1.7× higher LUTs and 3.2× higher flip-flops compared to a custom node decoder. When the custom node decoder is combined with a MicroBlaze-based controller, we observed 7.3×–16.6× higher latency for implementing the SMEM algorithm compared to a custom node decoder coupled with a custom controller implementation.

5.1.6.B.2 Processing Elements We now describe the different processing elements in the ERT seeding accelerator.

Index Fetcher: The Index Fetcher is responsible for initiating a walk by converting a k-mer string to an index table address and requesting the corresponding entry from the ERT index table. These requests immediately trigger a context switch, swapping out the current context until the requested data is returned. If the path terminates at the index table (entry type = `EMPTY`), the results are returned to the control processor to determine how to proceed. If the radix tree for that k-mer exists, the index fetcher issues a request for the root of the radix tree.

Tree Walker: The Tree Walker is responsible for traversing the ERT, decoding nodes, and reporting the end result of a walk. Each node in the tree is decoded using the corresponding base-pair in the read to calculate the next ERT node address. If the Tree Walker ever detects that it needs more of the ERT data structure to continue its traversal, it requests the data from the Data Fetcher and triggers a context switch. During decode, the Tree Walker computes the address of the next tree node based on the types and content of existing child nodes and the read characters or ends the traversal. Each ERT node takes a variable number of cycles to decode depending on node complexity. For example, `UNIFORM` nodes require an exact match string comparison to compare each DNA base-pair in the `UNIFORM` string with the read string. This comparison is accomplished using parallel XOR gates and priority encoders over three cycles. Leaf nodes that are "early path compressed" also require string comparison hardware (Section 5.1.5.B). Implementing these comparisons using custom parallel hardware is an important feature of the specialized processor versus implementation in software on a general purpose CPU.

Leaf Gatherer: If a tree walk hits an `EMPTY` node (i.e., match cannot be extended further) all remaining leaves in the parent sub-tree must be gathered in order to identify all possible reference locations of the current match. We refer to this as *Leaf Gathering*, and accomplish it using depth-first search (DFS) on the ERT sub-tree. This DFS is accomplished by considering and decoding each base-pair (A,T,G,C) path in the ERT and maintaining a stack of ERT node indices that need to be explored. Nodes are decoded and traversed just as in the Tree Walker, however, the Leaf Gatherer does not need to perform string matching (required for early path compression and `UNIFORM` nodes), and does not include string comparison hardware.

5.1.6.B.3 Control Processor The SMEM search algorithm consists of several input-dependent conditional branches that are hard to predict in general-purpose processors. Our control processor overcomes this by implementing the high-level algorithm for SMEM search in hardware. For example, if a forward walk finishes, the control processor looks at the start and end point, determines the condition of the finished walk, and issues a new command (e.g., get the leaves associated with the walk if the walk produced an SMEM, or start a new backward search if the walk failed to produce an SMEM) to the corresponding processing element command queue. To simplify tree walking hardware, walker PEs do not have separate hardware for forward or backward walks; the control processor issues a forward or backward walk command by providing a start index and the forward read (for forward searches) or reverse complemented read (for backward searches). The control processor maintains a queue of pending tree walks to deal with variable tree traversal times and schedules walks from other reads to ensure good compute utilization. Our accelerator is designed to be flexible enough to also implement other algorithms based on the FMD-index. This would require adding new control FSMs to the Control Processor, while all other hardware structures (index fetchers, tree walkers, leaf gatherers, crossbar, and I/O) can be reused.

5.1.6.B.4 K-mer Reuse Metadata Storage and Sorting In order to perform k-mer reuse (Section 5.1.5.I, Figure 5.10), all backward search LEPs for a forward match in a read must be exported to the k-mer metadata table. Backward searches that share the same k-mer are grouped together using the parallel hardware sorter [144] to group entries for the same k-mer (Phase 2 in Figure 5.10). We also implement a specially designed cache structure–*the k-mer reuse cache*–to cache index table lookups, ERT root node accesses, and other ERT accesses. It is sized conservatively using high-coverage human reads (details in Section 5.1.6.B.6, Table 5.3, batch size = 1000), taking into account potential high-reuse use-cases e.g., high coverage input reads and reads from other repetitive genomes like wheat. We saw little reuse benefit from increasing batch size beyond 1000 and reuse cache size beyond 4 MB. The k-mer reuse cache is also direct-mapped. We settled on a direct mapped cache, since the observed hit rate was within 1.2% of a fully-associative cache. Since k-mer reuse forces the algorithm to generate MEMs out-of-order for a particular read, we must also store all MEMs for each read in intermediate on-chip storage, to perform MEM containment checks and finally produce SMEMs in a final reconciliation step.

5.1.6.B.5 System Integration and Programming API In this section, we describe the programming interface provided by the ERT seeding accelerator and its integration with the AWS EC2 F1 shell interfaces.

Host-Accelerator Interface: We adopt a system configuration similar to that in AWS EC2 F1, with both the host and the accelerator having their own physical memories (DRAM). We assume the existence of two communication channels from the host to the accelerator, similar to the AWS-F1 shell interface; one for data transfer to/from the accelerator custom logic (CL), for example, using 512-bit AXI-4 DMA transactions (XDMA) on PCIe Gen3 $\times 16$ links; and another for issuing control commands and accessing memory-mapped status registers using an interface such as the 32-bit AXI4-Lite interface (OCL).

Listing 5.1 shows the C API for programming our accelerator which builds on the AWS FPGA management libraries. The libraries include APIs for reading/writing large chunks of data to the accelerator via PCIe DMA and handling interrupts. Memory-mapped registers on the accelerator are used for configuration and status monitoring. We also implement overflow handling in the accelerator for reads with too many SMEMs that overflow the on-chip MEM result buffers. Our accelerator flushes these results into a designated overflow region in the accelerator DRAM to be later processed in the host.

```
/* Create batch of 2-bit encoded reads in host */
void encode(uint8_t* h_buf, char** reads, int* len);
/* PCIe DMA transfer of 'sz' bytes from host buffer
   to accelerator memory at offset d_off */
int writeData(uint8_t* h_buf, int sz, int d_off);
/* Write configuration register
   on accelerator to begin processing batch */
int startCompute();
/* Read status register
   on accelerator to determine completion */
int waitForFinish();
/* Read 'sz' bytes from offset d_off
   in accelerator memory to host buffer o_buf */
int readResult(uint8_t* o_buf, int d_off, int sz);
```

Listing 5.1: C Programming API for accelerator

Runtime System: We extend the multi-threading model in BWA-MEM to provide a separate worker thread, one each for managing our seeding and optional seed-extension accelerators interfaced over PCIe. These worker threads communicate via non-blocking producer-consumer queues.

The main CPU thread first allocates a buffer for the ERT-index and copies it to the accelerators' DRAM using XDMA transactions. Reads are then pre-processed in a worker-CPU thread which allocates a 64-byte buffer for each read, encodes each base using 2-bits (reads with ambiguous bases such as N are processed on the host) and copies the buffer to the accelerators DRAM. This thread also acquires a lock to one of the accelerator status registers using the OCL interface and signals the accelerator to begin computation. It continues to monitor the status of this register, till the accelerator updates it with a *done_seeding* command. At this point, another CPU thread retrieves SMEMs from the accelerator DRAM using XDMA transactions and processes any result-buffer overflows. These SMEMs pass through a chaining step and can optionally be processed similarly using a seed-extension accelerator.

We implement double buffering on our accelerator, so that memory transfers to/from the accelerator over PCIe can be overlapped with computation. Seeding results are encoded in the same format as the baseline prior to chaining (i.e., (seed start position in read, seed length, list of seed hits in the reference genome)) to eliminate overheads due to additional data structures for format conversion.

5.1.6.B.6 Evaluation Methodology ERT was built using the latest build of the reference human genome assembly (GRCh38) from the UCSC genome browser [34]. Decoy contigs and mitochondrial DNA are filtered out and only chromosomes 1-22, X and Y are used to build the ERT index. For the input reads, we choose the single-ended Illumina Platinum Genomes benchmark dataset (`ERR194147_1.fastq`) [67] consisting of 787,265,109 reads of 101 `bp` length also used in prior work [73]. Reads containing ambiguous base pairs (non-A/C/G/T) are processed on the host-CPU and ambiguous base pairs in the reference genome are converted to one of the standard nucleotides (A/C/G/T) using the same procedure as [120, 118].

Experimental Setup: We compare the software and FPGA/ASIC versions of ERT against BWA-MEM (v0.7.17 release) and BWA-MEM2 (commit `ebc2378`) (refer Table 5.2). ERT-PM adds the prefix-merging optimization while ERT-KR includes both prefix-merging and k-mer reuse. All software comparisons were performed on one of the best-available CPU instances from AWS EC2, `c5n.18xlarge` running 72 threads. BWA-MEM, BWA-MEM2 and software ERT scale well with thread count, given sufficient memory bandwidth. The detailed system configuration is shown in Table 5.1. CPU power was estimated using Intel's RAPL interface. In software, the k-mer reuse optimization resulted in a 1.2% slowdown over prefix-merged ERT. This comes from the overheads of sorting k-mers prior to backward search, maintaining backward searches by k-mer in a metadata table and querying the software managed k-mer reuse cache on each index table/tree access. All reported results consider the three stages of seeding computation in BWA-MEM2: SMEM generation, reseeding, and LAST. We verified that our implementation produces identical seeds as BWA-MEM2 for the complete Illumina Platinum genomes dataset. To estimate the performance of different configurations of ASIC-ERT, we developed a cycle-accurate model using our software implementation and generated memory traces from the corresponding software runs for a representative set of 1 million reads from ERR194147, containing ∼80 % perfect matching reads and ∼20 % non-perfect matching reads similar to the full ERR194147 dataset. Ramulator [102] (commit `7ce65d`) was used to estimate performance and DRAMPower [50] (commit `6c5ebe`) was used to estimate DRAM power and energy.

c5n.18xlarge (AWS EC2 instance)	Intel Xeon Platinum 8124M 3 GHz; 2 sockets; 36 cores; 72 threads
L1 I&D cache	18 x 32KB Instruction; 18 x 32KB Data
L2 cache	18 x 1MB
L3 cache	18 x 1.375MB
Memory	192 GB DRAM

Table 5.1: Baseline system configuration.

Configuration	**Description**
CPU-BWA-MEM	Baseline BWA-MEM: 72 threads
CPU-BWA-MEM2	Baseline BWA-MEM2: 72 threads
CPU-ERT	Best configuration of ERT: 72 threads
ERT	Baseline ERT
ERT-PM	ERT with prefix merging
ERT-KR	ERT with prefix merging and k-mer reuse

Table 5.2: Seeding – comparison candidates for evaluation.

ASIC Configuration, Synthesis, and Frequency: Our RTL model for the seeding accelerator was synthesized using Synopsis Design Compiler 2018.2, HPC 28nm process, LVT standard cell library and 12t cells, at 1V (Table 5.3). The seeding processor achieves a 1.38 GHz clock frequency, and is limited by the operating frequency of SRAMs used for context memories. Each SRAM structure in our ASIC was compiled separately: considering word size, number of words, single/dual port requirement. TSMC's 28nm memory compiler is used for power/area estimation.

Component	**Configuration**	**SRAM (total)**	**Area (mm^2)**	**Power (mW)**
Seeding Machines Total	16×	2.72 MB	9.598	11,768.38
K-mer Sorter + Metadata Table	1×	8.26 MB	14.94	9,593.87
K-mer Reuse Cache	1×	4.02 MB	6.99	1,526.76
Seeding Accelerator Total	—	—	31.53	22,889.01
DRAM Power	8 channels	—	NA	2,185.7
Total System Power	—	—	—	25,074.71

Table 5.3: ASIC Configuration and Synthesis Results.

Component	**Configuration**	**LUT (%)**	**BRAM (%)**	**URAM (%)**
Index FU	1 × 8	0.32	0	0
Walker FU	3 × 8	13.76	0	0
Leaf Gathering FU	2 × 8	3.36	0	0
Command Queues	0.72 KB x 8	1.92	6.08	0
Context Memories	17.6 KB x 8	0	15.04	3.28
Control Processors	1 × 8	0.56	0	0
Data Fetcher	1 × 8	3.68	0	0
SMEM Result Buffer	2.3 KB x 8	0	0	13.28
MISC.		1.12	0	0
Seeding Machines Total	1 × 8	24.72	21.12	16.56
K-mer Sorter	—	1.95	0.3	26.77
K-mer Reuse Cache	4.01 MB	10.04	5	18.33
Seeding Accelerator Total	1	36.71	26.42	61.66
AWS Shell	–	19.74	12.63	12.20
Total	–	56.45	39.05	73.86

Table 5.4: ERT accelerator – Per-FPGA configuration and synthesis results.

FPGA Prototype: We prototyped and verified our seeding accelerator on Amazon's EC2 F1 FPGA cloud environment. We chose the `f1.4xlarge` instance with 2 FPGAs and equivalent bandwidth as the CPU configuration (64 GB/s peak bandwidth per FPGA). Each FPGA in the F1 instance is a Xilinx XCVU9P with 2,586K logic cells, 36.1 Mbits of Block RAM and 270 Mbits of UltraRAM. The accelerator is implemented in System Verilog, placed-and-routed at 250 MHz. System configuration and synthesis results along with the overheads of the AWS Shell interface are shown in Table 5.4.

5.1.6.B.7 Results Figure 5.16 shows the performance of the seeding step expressed as Million reads/s across different configurations. It can be seen that the software version of ERT provides 2.1× speedup over the state-of-the-art BWA-MEM2 baseline running on 72 threads. This is because ERT greatly reduces the amount of data fetched per read leveraging multi-character lookup and optimizations for spatial locality.

Overall, ASIC-ERT achieves 8.1× improvement in seeding throughput over multi-threaded BWA-MEM2. ASIC-ERT-Baseline utilizes 256 contexts to saturate memory bandwidth and achieves a 2.05× throughput improvement over the CPU-version of ERT. Using prefix-merged radix trees, allows us to reduce the number of backward extensions and further improve throughput by 1.23×. By leveraging temporal locality in the backward search pass, k-mer reuse further improves the overall seeding throughput by 1.56×.

FPGA-ERT achieves a throughput of 3.6 Million reads/s resulting in a speedup of 3.3× over baseline CPU BWA-MEM2. Our FPGA-ERT prototype inherits some limitations of the AWS FPGA-memory interface for ERT-style accesses, not present in the ASIC configurations. For instance, we could not customize the 3^{rd}-party memory-controller IP to return subsequent memory requests to the same DRAM page with lower latency, unless AXI burst transactions with large burst lengths (>64 B) were used. However, always using large burst-length transactions for ERT accesses leads to data wastage. Also, large burst lengths increase datapath complexity and on-chip storage on the FPGA. By issuing 128 B requests when possible, we observed ∼5–8 GB/s per-channel for ERT accesses, although peak channel-bandwidth is 17 GB/s.

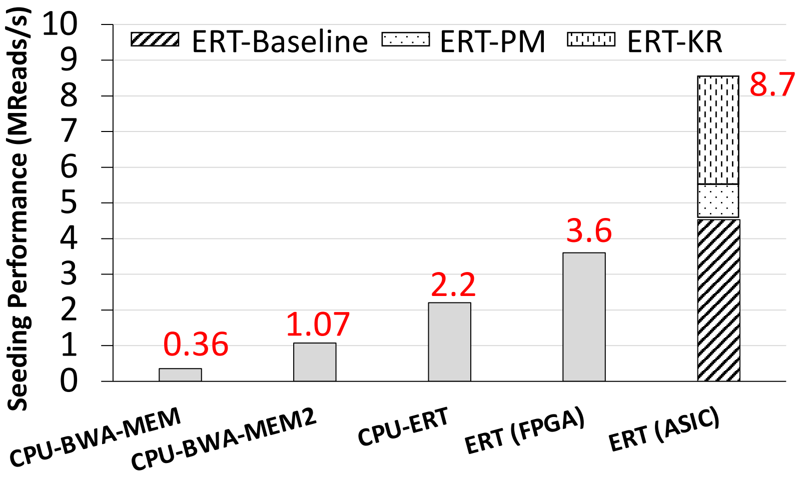


Figure 5.16: Seeding performance in million reads/s.

Memory Access Characteristics: To further understand the reasons for improvement in seeding throughput, we discuss the memory access characteristics of different configurations. Figure 5.17 shows the average number of memory requests and the data fetched for BWA-MEM and the different configurations of ERT. Compared to BWA-MEM (BWA-MEM2), ERT makes 6.7× (4.5×) fewer memory requests per read. This is because ERT nodes are tightly packed into cache

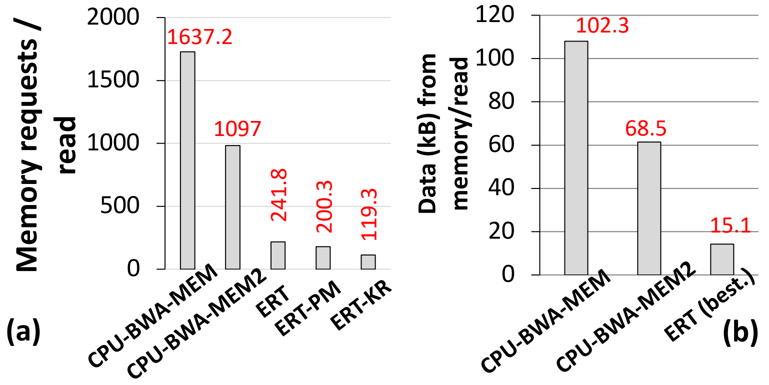


Figure 5.17: Memory access characteristics of different designs.
(a) Memory requests per read (b) Data requirements per read (in KB)

lines to improve spatial locality. On average, ∼3 ERT nodes are traversed per 64 B, utilizing 50% of the data. Also, ERT-KR leverages the k-mer reuse cache to further reduce the number of memory requests by ∼2×. This leads to low data requirements per read (15.1 KB).

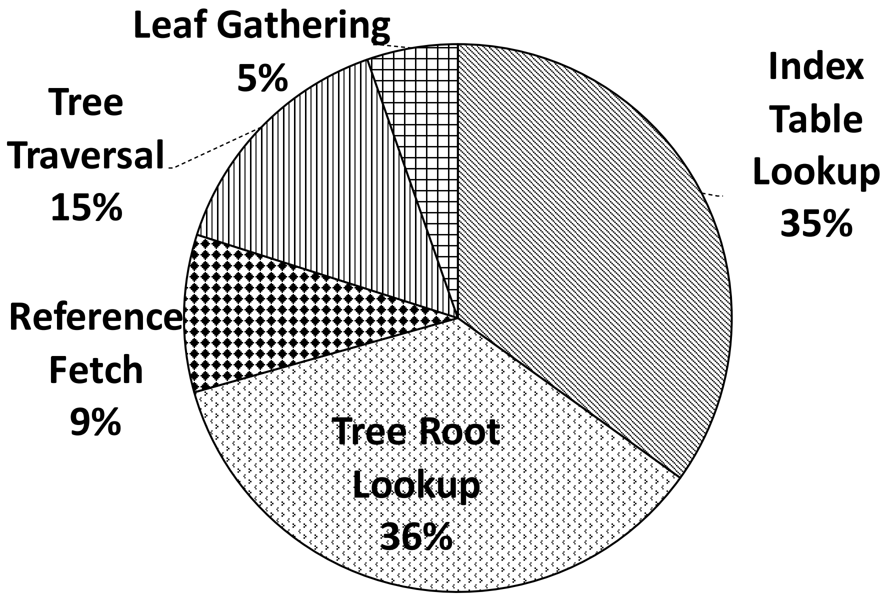


Figure 5.18: DRAM page open breakdown for ERT-KR

Seeding Performance Breakdown and Efficiency: Figure 5.18 shows the distribution of DRAM page opens in ERT for the different steps in seeding. Tree traversal and leaf gathering only contribute to a small number of page opens indicating high spatial locality in these steps (15% and 5% respectively). Furthermore, the multi-level index table in ERT reduces the number of node traversals by allowing multi-character lookups. In the baseline ERT, the penalty for index table and radix tree root lookup must be paid for almost every k-mer. Since these accesses are random, they contribute to 71% of the DRAM row buffer misses.

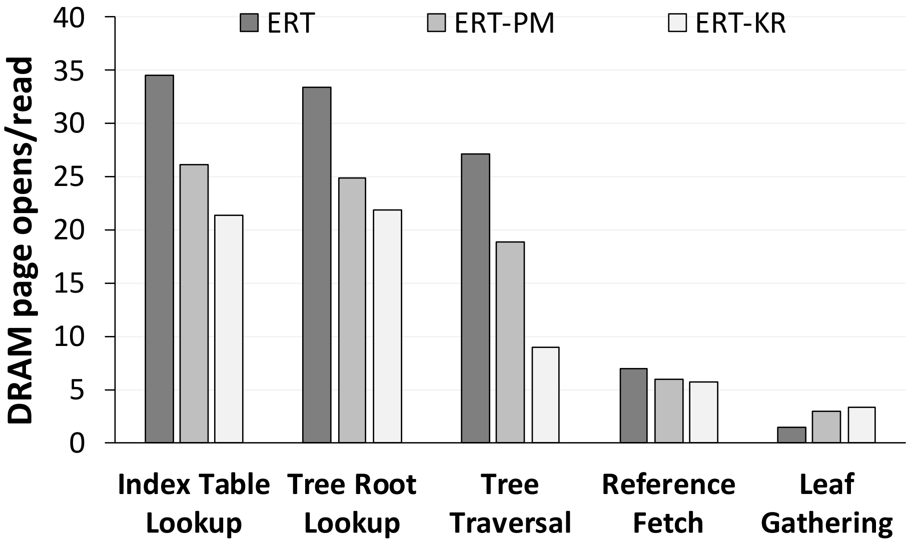


Figure 5.19: DRAM page opens per read across optimizations.

Figure 5.19 shows how prefix-merging and k-mer reuse can be leveraged to reduce the number of row buffer misses in each of these steps. Prefix-merged radix trees reduce the work done during backward search and reduce index table lookups by 24.4%, tree root lookups by 25.5% and tree traversal by 30.4%. In addition, k-mer reuse amortizes the cost of backward searches across several k-mers and leverages temporal locality using the k-mer reuse cache to reduce index table lookups by 37.9%, tree root lookups by 34.3% and tree traversal by 66.7% compared to baseline ERT. Note that reference fetch to obtain the complete strings stored at leaf nodes accounts for 9% and incurs nearly the same cost for all three configurations. Since k-mer reuse does not impose the right-to-left order for the backward extensions of a given read it cannot take advantage of the early termination of backward searches as described in (Section 5.1.5.L). This results in a slight increase in DRAM page opens for leaf gathering over baseline ERT.

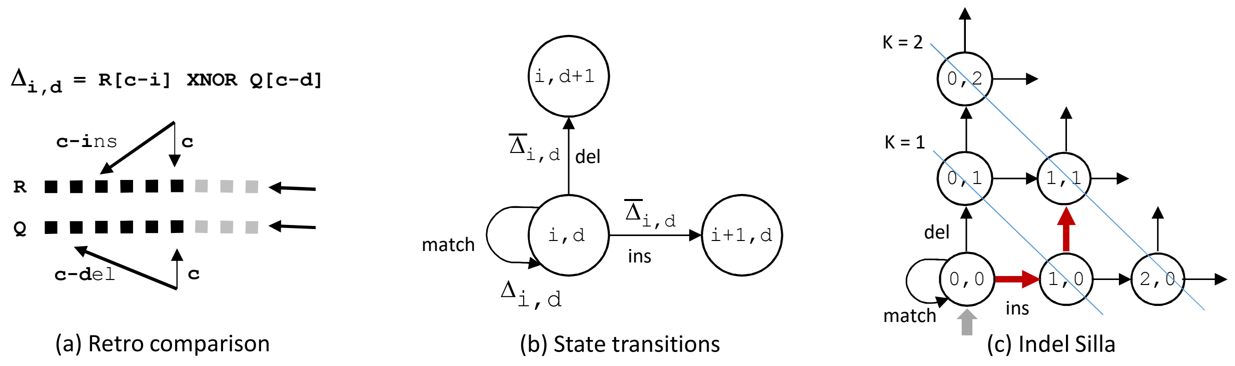


Figure 5.20: Silla design.
Retro comparisons (a) are used to determine state transitions (b) in Indel Silla (c).

5.2 Approximate String Matching Acceleration (Seed-Extension)

The previous section discussed a common exact string matching problem in genomics, i.e., seeding. This section describes a computationally intensive approximate string matching kernel called seed-extension and our approach to accelerate this kernel.

In particular, we describe the design of a non-deterministic finite state automata for approximate string matching called String Independent Local Levenshtein Automata (Silla). Silla is designed from the ground-up to enable efficient hardware acceleration. Silla solves the following problem: Given two strings, a reference `R` and a query `Q`, compute the minimum Levenshtein (edit) distance between them if it is less than a small bound `K`.

5.2.1 Silla for Indel

We first describe the Silla design assuming only insertions and deletions (indels), and then extend it to support substitutions. A key observation is that we can use the states to represent the number and type of edits made so far, and not explicitly track the matches as it is done in Levenshtein automata. Figure 5.20 (c) illustrates indel Silla for a maximum edit distance of two (`K = 2`), where a state `i,d` means that when that state is reached, the automata has seen `i` insertions and `d` deletions. All states have a match transition back to the state itself as shown for the start state (omitted for other states for clarity).

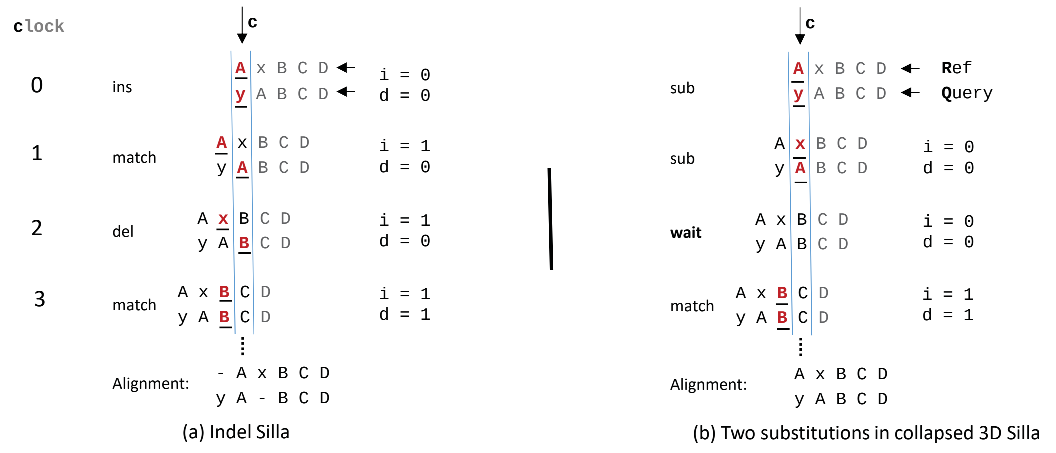


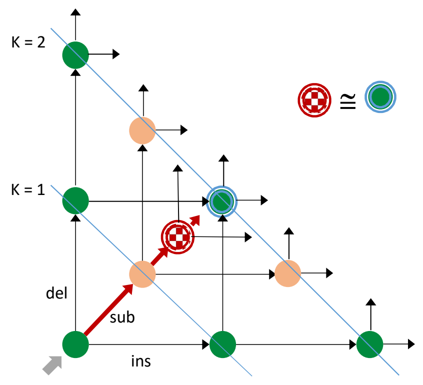
Figure 5.21: Silla illustration.

Computation begins at the start state (`0,0`) which represents no edits (`i=d=0`). At all active states, one character from each of the two strings is compared in every cycle (step) starting from the first character. As long as there are no edits, the positions of the compared characters in the two strings `R` and `Q` are the cycle number `c` as shown in Figure 5.20(a) by the two vertical arrows labeled `c`. We refer to this position `c` as the *cycle* position.

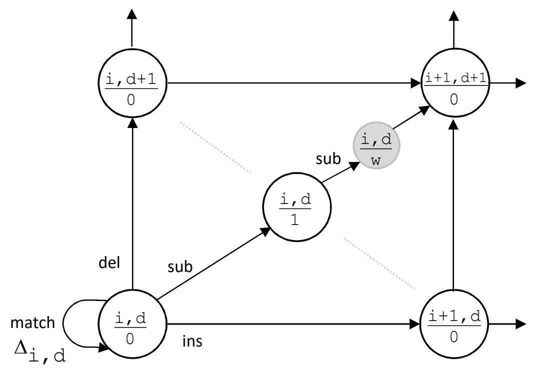
On an insertion (or a deletion) the position in the reference (or query) needs to be offset with respect to the cycle position. This can be understood from the example in Figure 5.21 (a). As the comparison fails in the first cycle (`A` $\neq$ `y`), Silla explores as one possibility that a character (`y`) is inserted into the query `Q` by transitioning into state `1,0` . In the next cycle, state `1,0` should compare the previously unmatched character `A` in the reference `R` to the current character in the query `Q`. This is achieved by offsetting the character position in the reference `R` by as many insertions as the state represents (one in our example).

Similarly, the character position in the query `Q` is offset by the number of deletions. For example, when the comparison fails again in the third cycle (`x` $\neq$ `B`), Silla explores deleting `x` from `R` by transitioning to state `1,1` . The new state increments the character position offset for the query, so that the unmatched character `B` from the previous cycle is again compared, but this time to the following (now current) character `B` in the reference.

Silla then generates two more matches for characters `C` and `D` and thus discovers a solution



(a) 3D Silla



(b) State transitions in collapsed 3D Silla

Figure 5.22: Collapsing 3D Silla to two dimensions.
Silla now supports indels and substitutions in two dimensions.

for aligning the given two strings within an edit distance of two. The final alignment is shown at the bottom of the Figure 5.21 (a).

Thus, the character positions whose comparison controls a state is determined by the indels that the state represents. We refer to these comparisons as *retro comparisons*, and the offsets as *indel offsets*. Figure 5.20 (a) shows the equation and its illustration for computing the retro comparison for a state `i,d`. The state transitions based on a retro comparison is depicted in Figure 5.20 (b). As you can notice, Silla explores both options, insertion and deletion, when a retro comparison fails for a state.

All the states in Silla are accepting states. After Silla completes processing of a pair of strings, the remaining active states represent possible string alignments with edit distance `i+d <= K`. The active state with the smallest indel indicates the minimum edit distance for the given strings. If no states remain active at the end of processing, there is no alignment with `indel < K`. The number of states in the indel Silla is $(K + 1) * (K + 2)/2$.

5.2.2 3D Silla for Substitutions

We now extend Silla to support substitutions. An easy solution for tracking substitutions is to add states in the third dimension to Silla. Each layer in the third dimension looks like a 2D indel Silla, and there are as many layers as the maximum possible number of substitutions (which is limited by `K`). When the retro comparison fails at a state `i,d|s`, to explore substitution, Silla transitions to a corresponding state in the next substitute layer along the third dimension (`i,d|s+1`).

Figure 5.22 (a) depicts 3D Silla. A state's color represents the 3D layer it belongs to. As there are `K+1` layers, we have $(K+1)^2/2*(K+2)$ states.

5.2.3 Collapsed 3D Silla for Indels and Substitutions

3D Silla requires $\mathcal{O}(K^3)$ states. Furthermore, a hardware for 3D Silla would also be inefficient due to challenges in laying out a 3D design on a 2D plane. We avoid these problems by reducing a 3D Silla to an equivalent 2D Silla as follows.

Our key observation is that we need only one additional layer of 2D Silla, not K, to support substitutions, and that we can collapse the states needed in the higher substitution layers into one of those two layers.

Intuitively, the reason for having two dimensions in the 2D indel Silla is that we need to track the indel offsets in the two strings for different states of the automaton. However, a substitute action does not change the indel offsets and the function of the third dimension in the 3D Silla is simply to ”count” or record the number of substitutions. Since we are only interested in the total edit distance, we notice that state `i,d|s` in the 3D Silla has the same edit distance as state `i+1,d+1|s-2`. Furthermore, the *relative* indel offsets of these two states is also the same `i-d`, although state `i+1,d+1|s-2` is shifted one character position earlier in the string than `i,d|s`. Hence, we can merge state `i,d|s` with state `i+1,d+1|s-2` by inserting one wait cycle in the path from `i,d|s-1` to `i+1,d+1|s-2`.

The example in Figure 5.21 (b) illustrates this merger operation. It is for the same two strings discussed before, but this time we discuss a solution that uses two substitutions to align them instead of an insert followed by a delete. When retro comparison fails in cycle 0 (`A` $\neq$ `y`), control switches to the `0,0|1` state to explore a substitution. When the comparison fails again (`x` $\neq$ `A`), to explore another substitution, 3D Silla would transition to `0,0|2`. But, as noted above, the number of edits represented by the state `0,0|2` is same as `1,1|0` and the relative difference between their indel offsets is the same (zero). Hence, in cycle 2, a `0,0|2` state would be comparing the two characters `B`, which are the same characters as state `1,1|0` is comparing in cycle 3. In a way, state `0,0|2` is one cycle *ahead* of `1,1|0`.

Therefore, we can merge `0,0|2` with `1,1|0` simply by delaying the path from `0,0|1` to `1,1|0` on substitution by one cycle. Figure 5.21 (b) illustrates this. When the retro comparison fails in cycle 1, Silla transitions to a wait state that takes no action in cycle 2. In the next cycle 3, the execution correctly resumes in state `1,1|0`.

To generalize, our final Silla design supports indels and substitutions using two layers of 2D Silla. Final state transitions are shown in Figure 5.22 (b). Matching transitions are again omitted for clarity. To explore a substitution from a state `i,d|1` in the second layer, Silla transitions to a wait state `i,d|w`, and then in the following cycle, transitions back to merge with a state in the first layer `i+1,d+1|0`. Figure 5.22 (a) can now be re-interpreted as a collapsed 3D Silla, where the checkered states represents the wait state and has a single outgoing transition to merge with states in the first layer.

Collapsed 3D Silla has $(K+1)*(K+2)/2$ regular states in each of the two layers, and also has an additional $(K+1)*(K+2)/2$ wait states resulting in a total $3(K+1)*(K+2)/2$ number of states. Also, by grouping states `i,d|0`, `i,d|1` and `i,d|w` together as one unit in the layout, a completely regular design is obtained with only local communication between neighboring units. Henceforth, we refer to this collapsed 3D Silla simply as Silla.

5.2.4 Merging Confluence Paths is Sound

Silla explores multiple solutions concurrently in different states. When a retro comparison fails, a state activates all three of its outgoing edges to pursue all possible edits to handle the mismatch. For example, Figure 5.21 shows two paths for the same string. However, in fact, Silla would explore many more paths than shown, and often there are more than one solution. In the example, the path with a deletion and an insertion and the path with two substitutions are both optimal solutions.

Silla merges a set of paths that reach a state in the *same* cycle into one active path. We refer to these paths as confluence paths. A state in Silla can have as many as four incoming edges (e.g., state `1,1|0` in Figure 5.22 (a)), including the matching edge.

Fortunately, it turns out that merging confluence paths is safe without additional precautions. The reason is as follows. For a given Silla state, and given cycle, we can partition the reference (`R`) and query (`Q`) strings as `x||y` and `u||v`, where `x` and `u` are prefixes of `R` and `Q`, respectively. A nice property is that this partition is the same for all the confluence paths. The prefixes that have been processed in the previous cycles, will not be examined going forward, and it is guaranteed that all the confluence paths observed the same number of edits for them. All the confluence paths also share the same suffix, and the edit distance computation for the unprocessed suffixes (`y` and `v` in `R` and `Q` respectively), is *independent* of the path taken so far. Thus, it is not necessary to independently explore that same suffix for each of the confluence paths, and therefore they can be safely merged.

5.2.5 Silla Accelerator for Genome Sequencing

This section presents the Silla hardware accelerator (SillaX) for genomics. It uses a form of a systolic array architecture that efficiently computes and locally distributes retro comparisons to all the states. Besides edit distance, it also supports more sophisticated scoring schemes based on the affine gap penalty [77] used in genomics. Finally, it adds capability to traceback the sequence of edits made to reach the final alignment solution. These capabilities are essential to perform seed-extension in genome sequence alignment.

We synthesized and validated the implementation for a whole human genome and confirmed that its output matches that of Broad Institute's BWA-MEM standard pipeline [118] for all 787,265,109 single-ended reads.

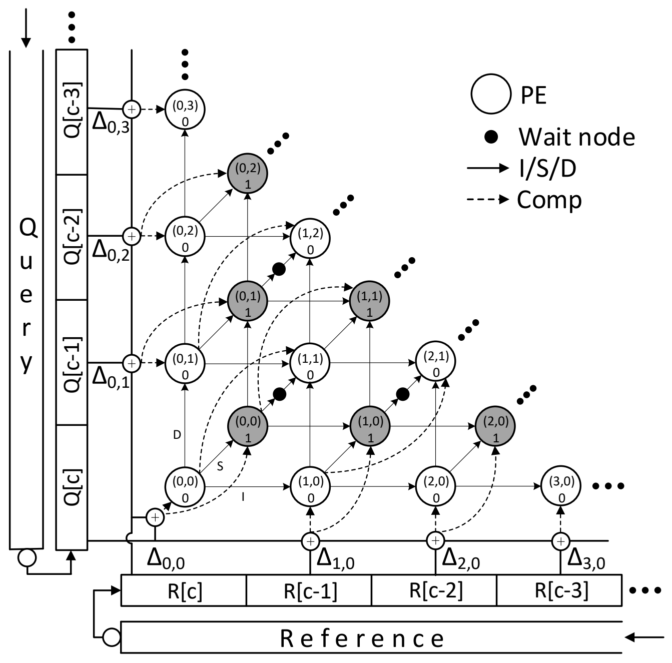


Figure 5.23: SillaX accelerator.

5.2.5.A SillaX Edit Machine

SillaX implements each regular state in Silla as a small processing element (PE) shown in Figure 5.24. PEs for wait states are not shown, but they simply activate the outgoing edge when they are active. An important property that Silla guarantees is that a state has to communicate only with its neighbors. This allows us to connect all the PEs using a locally communicating regular network (Figure 5.23). We refer to a PE simply as a state in our discussions.

Every regular state is controlled by the result of its retro comparison in each cycle. A significant challenge that we address is the efficient calculation and distribution of the retro comparisons to all the regular states. In a naive system, we would need as many retro comparisons as the number of regular states ($(K + 1)^2/2$), every cycle. However, across two clock cycles, many of the the retro comparisons are reused. This allows us to solve this problem with just $2K + 1$ comparisons per cycle as described next.

A retro comparison for a state is computed based on the current cycle (c) and that state's indel

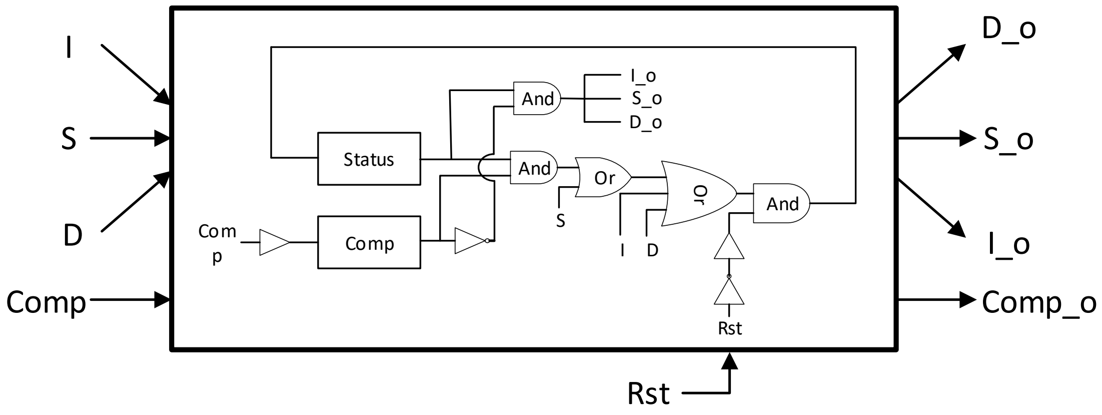


Figure 5.24: PE for SillaX edit distance machine.

(`i,d`) as we discussed in Figure 5.20(a). *We observe that the states along a diagonal can reuse the retro comparisons.* A state `i,d` needs the same retro comparison that `i-1,d-1` needed a cycle earlier. Therefore, in each cycle, SillaX computes the retro comparisons for all the peripheral states, $\forall i$ `i,0` and $\forall d$ `0,d`, and then shifts them diagonally into the interior states every cycle. That is, a state `i,d` latches its incoming retro comparison and forwards it to `i+1, d+1` the next cycle (`Comp` in Figure 5.24).

To implement the above functionality, SillaX has two sets of shift registers, one for each dimension. Input characters from two strings `R` and `Q` flow through those shift registers as shown in Figure 5.23. A set of $2K + 1$ comparators outside the grid compute the retro comparisons every cycle ($K + 1$ comparisons for each dimension with one common comparison for `0,0`). Note that computing and distributing the retro comparisons again requires only local communication between neighboring states allowing for a highly scalable design. The only exception to this local communication is the distribution of the current cycle's characters in the reference and query strings `R[c] and Q[c]` which are distributed across the entire periphery. However, distribution of these two values can be accomplished using a distribution tree which delivers the values to all comparators at the same time, much like a clock tree distributes a synchronized clock to all the PEs.

Efficiency: SillaX requires only $\mathcal{O}(K^2)$ states (processing elements) and computes in about N cycles, where K is the edit distance and N is the string length. Typically, $K << N$. Our

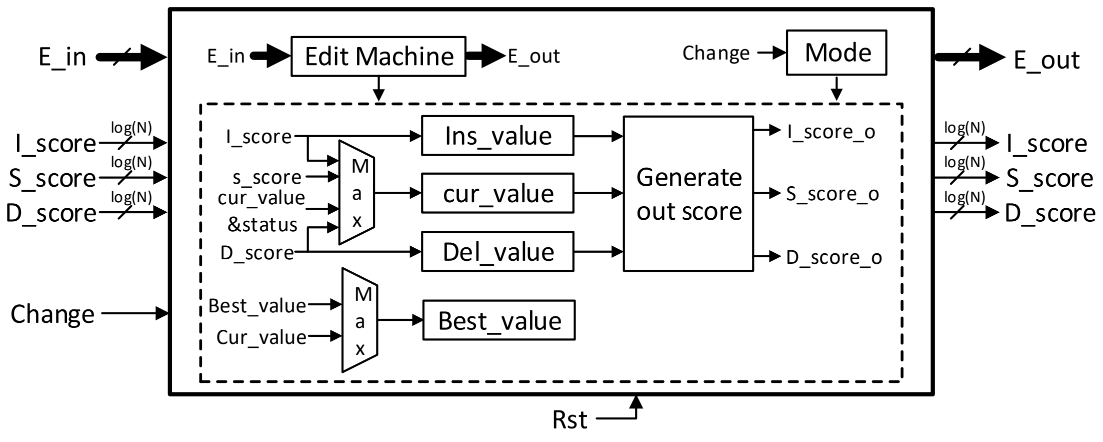


Figure 5.25: PE for SillaX affine gap scoring machine.

implementation in 28nm can be clocked at 6 GHz, and each PE has only 13 gates. This is significantly more efficient than software implementations, whose time complexity is $\mathcal{O}(N^2)$. Hardware accelerators for these require $\mathcal{O}(N)$ processing elements, which does not scale as well for large N.

5.2.5.B Scoring Machine

Edit distance is a simple form of scoring alignment between two strings which can have many different uses. However, read alignment in genome sequencing uses a more sophisticated scoring scheme based on empirical evidence gathered from analyzing many genomes [134, 77].

If we use a constant score for each type of edit, then it remains safe to merge *confluence paths* by selecting the one with the highest score, as the properties discussed in Section 5.2.4 continue to hold.

The scoring scheme used in the standard BWA-MEM pipeline, however, raises a new problem. It rewards every match (+1) and penalizes every substitution (spenalty = −4) with predefined scores. Each indel, which represents a set of consecutive deletions or insertions, is penalized using affine gap penalty (G):

$$G = gopen + gextend * id,$$

where id is the number of characters deleted or inserted, gextend (−1) is the penalty for each consecutive edit, and gopen (−6) is an additional one-time penalty for each indel.

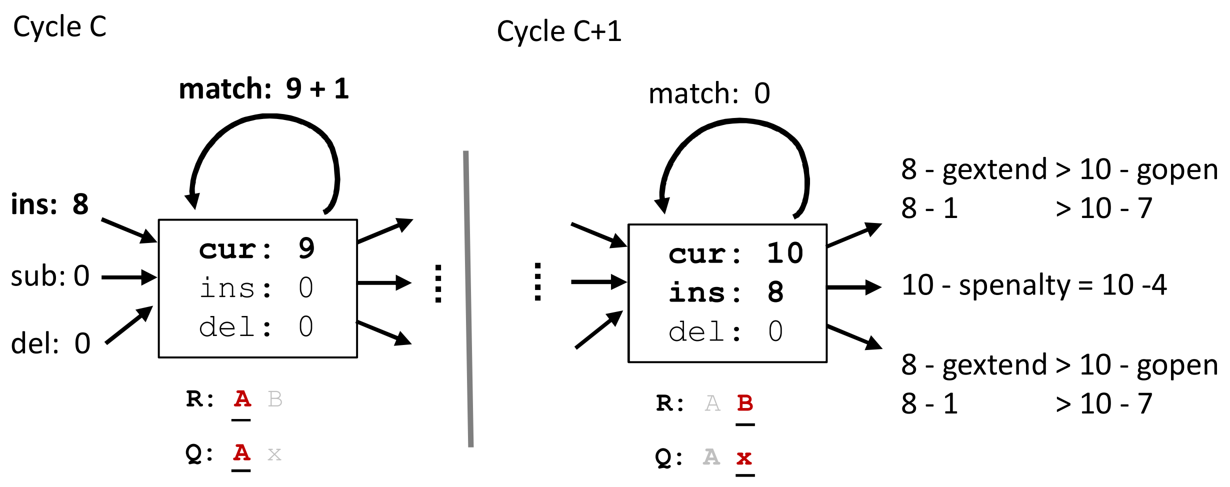


Figure 5.26: Delayed merging is needed to support affine gap penalty.

Given this, paths that have opened an indel gap (`open-path`) have an advantage over a path where the latest retro-compare is a match or substitution (`closed-path`). An open-path need not pay a gap opening penalty for the next insertion or deletion, but a closed-path should. As a result, we cannot merge the confluence paths into one at a given state and cycle based on their current scores alone since the future score depends on whether the confluence path is open or closed. Fortunately, we can address this by delaying the merge to the following cycle as shown in Figure 5.26.

To enable delayed merge, we latch the scores of the incoming active insertion and deletion paths in a state as shown in Figure 5.25. If there is a mismatch in the next cycle, we can select the best outgoing indel path by adding the gap penalty to previously closed paths. If there is a match in the next cycle, we can select the active path for the state by choosing between the closed paths and the open-paths from the previous cycle, which are now closed due to the match, based on which path has the highest score. We refer to this technique as *delayed merging*.

Closing an open-path due to a match in a state would prevent that open-path from potentially exploring a better solution in the higher edit states without having to re-open the path. For this reason, in the scoring machine, an active state conservatively activates the outgoing insertion and deletion transitions even on a match.

Figure 5.26 illustrates delayed merging. In cycle `C`, we cannot discard the incoming insertion open-path in favor of the matching closed-path, although the latter has a higher score. Instead, we latch its score. In the next cycle, when there is a mismatch, the latched open-path produces the best score for outgoing indel transitions.

After processing the strings, we need to compute the best score. BWA-MEM applies a heuristic called *clipping*, where it selects the best score seen during seed-extension, instead of determining the best score only among the final states at the end of string processing. The reason for this is the expectation that the ends of a read are likely to suffer from sequencing machine errors and are less likely to be true variants.

The SillaX scoring machine supports clipping as follows. Each state stores the best score it has seen during the entire computation. Once the strings are processed, the mode of the machine is changed to instruct the states to back-propagate their best scores, through local communication, but now in the reverse direction. Each state receives the best scores from three of its upstream nodes, computes the maximum of those scores along with its own best score, and passes the computed maximum to its downstream paths. In this way the best score seen at any time during the string processing in any state is read out at node `0,0|0`.

Efficiency: A scoring PE includes an edit PE, four scoring registers (`log(N)` bits each), and a programmable scoring logic. Three additional output and input ports are used for communicating scores. It also takes an additional K cycles to back-propagate the scores to the starting node. In spite of these additional overheads, the scoring machine remains efficient in space ($\mathcal{O}(K^2)$) and time ($\mathcal{O}(N)$).

5.2.5.C Traceback Machine

Read alignment requires the sequence of edits made and their positions in the string for the best solution found. This step is referred to as traceback. Hardware accelerators based on Smith-Waterman [154] typically delegate this step to software, which then becomes the bottleneck, or require hardware space that is proportional to the read length [54].

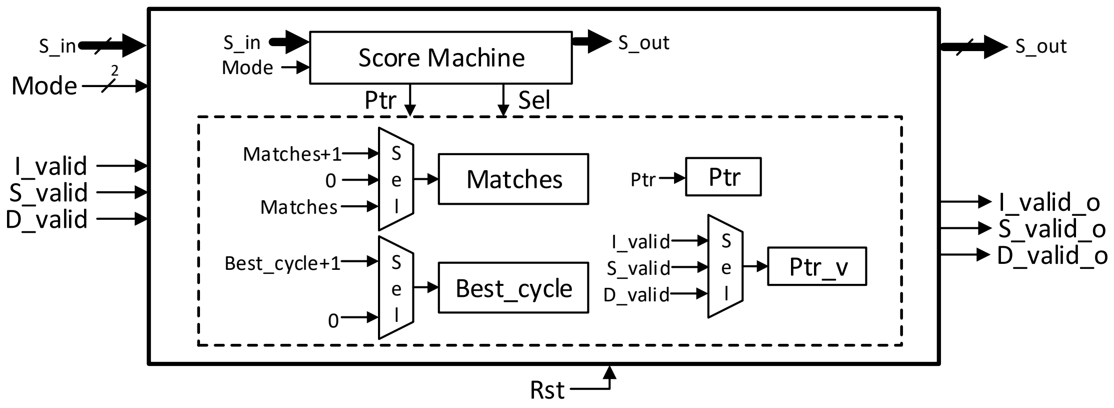


Figure 5.27: PE for SillaX affine gap machine with traceback.

We support traceback by extending our scoring machine as shown in Figure 5.27. Our main idea is to use a *pointer trail*, much like how an ant creates a scent trail to get back home. Also, we compress the trace representation in the machine by keeping a count of matches discovered in each state. That is, each state, in addition to tracking the best score it has seen, tracks the number of matches it found at that node for the path corresponding to the best score.

The traceback machine works as follows: In the string matching phase, when a state accepts a score from a downstream state as its best score, it sets its traceback pointer (`2 bits`) to that state. After processing the strings, in phase two, just like in the scoring machine, the best score is propagated back, along with the winning final state's identifier. In phase three, a signal is propagated forward to inform the winner. In phase four, by chasing the pointer trail from the winner, the states that are part of the winning path are flagged. In phase five, the trace is collected at the starting node `(0,0|0)` by shifting the matches and pointer values along the flagged winning path downstream, one state at a time per cycle.

One problem is that a pointer trail may get broken in phase one while processing the strings. When a *greedy* state discovers a higher score, it will discard the previous best score seen in an earlier cycle and its corresponding pointer. However, the previously discovered path that is now being explored in some upstream state may eventually emerge as the winner in the end. But the pointer trail for that winner is now overridden at the greedy state. To address this problem, we ask the greedy state to inform its upstream neighbors when it changes its pointer, which allows its neighbors to invalidate their pointer to that greedy state, thereby indicating that the pointer trail terminates in the neighbors state.

At the end of phase five, our controller examines the trace to check if it is complete. If it is broken, it re-runs the machine till the cycle when the winning path left the greedy state. We determine this cycle by keeping track of the cycle at which the best path seen in a state left that state. This is in addition to the best score that we had in the scoring machine. After the re-run, the machine collects the trace from the greedy state, which is now guaranteed to have the correct pointer. If the machine were to discover a new greedy state, it is resolved by re-running the machine again. In practice, however, we find that it is rare for pointer trails to be broken, and hence re-runs are rare.

Efficiency: Traceback machine has five phases, as opposed to two in the scoring machine. The first phase takes N cycles and the remaining phases take about K cycles each. It also adds a counter for tracking matches and a register for best cycle (each of size `log(N)`). These additional overheads are significantly lower than the $\mathcal{O}(N)$ space complexity in previous hardware accelerators that supported traceback [54].

5.2.5.D Composable SillaX

A key issue in hardware acceleration is maintaining flexibility so that a wide range of applications can be addressed. In our application this means that an accelerator should address both different string and edit lengths. Our proposed SillaX accelerator already allows arbitrary string length. However, the maximum edit distance is constrained by the the size of the PE grid and is fixed in hardware. To address this issue, we propose the use of composable sub-grids where multiple smaller SillaX engines can be combined into fewer larger engines or one maximum size engine. This creates flexibility where a few high edit distance machines can be reconfigured with a simple mode-switch into multiple smaller edit distance machines, thereby optimally addressing the targeted application space.

The concept is illustrated in Figure 5.28. Six small SillaX accelerator tiles are labeled by their

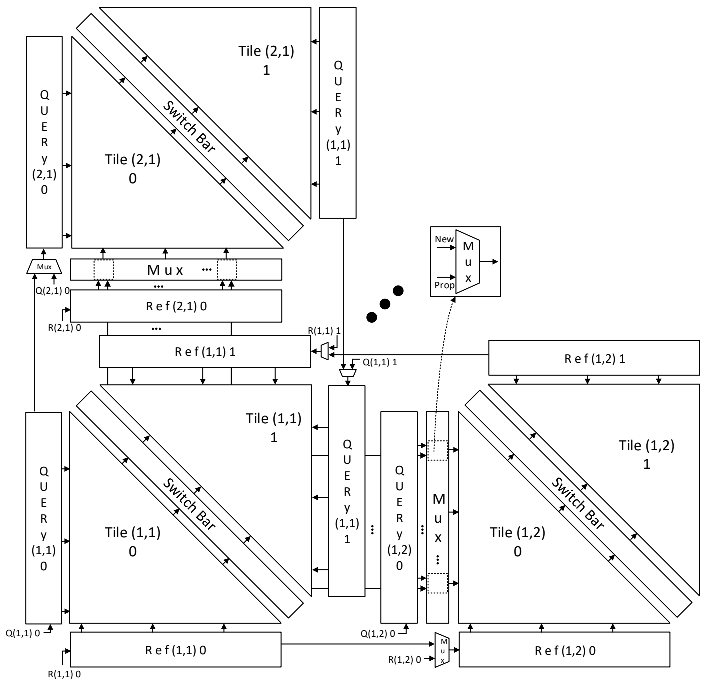


Figure 5.28: Illustration of composable SillaX.

position in the larger grid ((1,1), (1,2) and (2,1)) and whether they are oriented *forward* (0) or *flipped* (1). Each of these six accelerators can operate independently providing six engines each with the same edit distance K. Note that in the forward oriented tiles, state activation propagates from bottom left to top right as in the previous figures. In the flipped tile, state transitions propagate in the opposite direction. To make a SillaX accelerator with edit distance $2K$, we can combine the following four tiles: (1,1)—0, (1,1)—1, (1,2)—0, (2,1)—0 by changing the configuration of MUXes. In this case, the reference string will stream from Ref(1,1)—0 to Ref(1,2)—0, concatenating the two shift registers. Similarly, the two query registers are concatenated. Also, the connections inside Tile (1,1)—1 are reversed so that state transitions propagate from bottom left to top right. This is accomplished by adding MUXes/tri-state gates at the input/outputs of each PE. They configure which wires are treated as inputs and outputs in an array Tile. Finally, the outputs of Tile (1,1)—1 are fed into the inputs of Tile (1,2)—0 and (2,1)—0 forming a single larger array of PEs instead of 4 smaller ones. Note that in this example, Tile (1,2)—1 and Tile (2,1)—1 are still operating as independent SillaX engines with edit distance K.

This reconfiguration approach incurs only a small overhead of MUXes between tiles and for each PE. It allows many different configurations with edit distances ranging from K to pK where $p = sqrt(T)$ for an implementation with `T` Tiles. This broadens the application space of SillaX.

5.2.6 Comparison with Banded Smith-Waterman

Banded Smith-Waterman focuses on identifying near-exact matches (less than K edits) between genomic strings [84], similar to SillaX. Software-based banded Smith-Waterman implementations, however, have $\mathcal{O}(KN)$ time and space complexity. Hardware-based systolic implementations require $\mathcal{O}(N)$ time with $2K + 1$ processing elements and additional $\mathcal{O}(KN)$ space for traceback. In contrast, SillaX differs from prior banded Smith-Waterman implementations in the following ways.

Each PE in SillaX has 30$\times$ lower area than a banded Smith-Waterman PE when edit distance is used as the scoring scheme (300 um^2 vs 9.7 um^2 for SillaX at 5 GHz). Assuming a conservatively high K (=32) for aligning Illumina short reads, both SillaX edit machine and scoring machine achieve better area efficiency compared to banded Smith-Waterman because of fewer gates used in each PE. Furthermore, SillaX enables efficient in-place traceback within PEs. Hardware-based banded Smith-Waterman requires additional $\mathcal{O}(KN)$ space for traceback. Hirschberg's algorithm [88] reduces space to $\mathcal{O}(K)$, but increases time to $\mathcal{O}(NlogN)$. There exists no prior accelerator that supports traceback in $\mathcal{O}(K^2)$ space or lesser without sacrificing time complexity.

Since Silla is based on automata theory, it can be easily mapped to versatile automata processors supporting variable-width input symbols such as UDP [69] providing greater flexibility in implementation. From the algorithmic viewpoint, besides the fact that Silla is as an important successor to Levenshtein automata, it can also be easily extended to solve other important problems such as Longest Common Sequence problem and automatic spell correction, as well as ones in the bioinformatics domain [106].

CPU	Intel Xeon E5-2697 v3 2.6GHz; 2 sockets; 28 cores; 56 threads
L1 I&D cache	14 x 32KB Instruction; 14 x 32KB Data
L2 cache	14 x 256KB
L3 cache	1 x 35MB
Memory	120GB DRAM

GPU	Nvidia TITAN Xp 1.6GHz; 3840 CUDA cores
Shared L2 cache	3MB
Memory	64GB DRAM, GDDR5X

Table 5.5: Baseline system configuration.

5.2.7 Evaluation Methodology

Reference genome and input reads: We used the latest major release of human genome assembly (GRCh38) from the UCSC genome browser [34] and filtered out unmapped contigs and mitochondrial DNA. Only chromosomes 1-22, X and Y were used. For our evaluation, we use real human genome reads with 50$\times$ coverage from the Illumina platinum genomes [67] dataset. The dataset consists of the NA12878 human reference (single-end ERR194147_1.fastq) consisting of 787,265,109 reads of 101 `bp` length.

System Configuration: The detailed system configuration is shown in Table 5.5. To study SillaX's alignment throughput, we compare against major software implementations of the Smith-Waterman algorithm. We use the SeqAn library [66] as the CPU baseline, and SW# [108] as the GPU baseline.

Synthesis: We synthesized the SillaX accelerator using the Synopsys Design Compiler (DC) in a commercial 28nm process. We synthesized all three Silla machines: edit, scoring, and traceback, to obtain their area, power, and latency with respect to different clock frequency targets.

5.2.8 Results

Area, Frequency and Power Figure 5.29 shows the power and area for each processing element (PE) in the SillaX edit machine and traceback machine. The optimal design points are highlighted. Scoring machine is comparable to the traceback machine, so we omit it. 2 GHz is the inflection

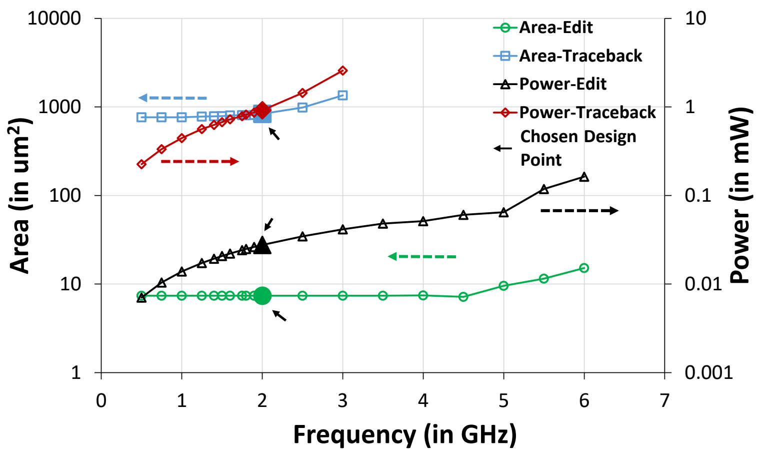


Figure 5.29: SillaX area and power for a single PE.

point. At 2 GHz, the SillaX edit machine has an area of 0.012 mm^2, power of 0.047 W and latency of 0.17 ns. At the same clock frequency, the traceback machine has an area of 1.41 mm^2, power of 1.54 W, and latency of 0.33 ns.

BWA-MEM reports alignments with score higher than 30. Using this estimate, we can derive that the edit distance (`K`) should be less than 32. Given this, we conservatively use `K = 40` in our analysis. To support `K = 40`, SillaX uses 1,681 processing elements (PEs).

GRCh38 Human Genome Assembly Validation: To evaluate the accuracy of the SillaX traceback machine, we ran all the the non-exact matching reads in the `ERR194147_1.fastq` file and compared the alignments produced with that from BWA-MEM. Exact matching reads are trivially identified using a single state SillaX machine that transitions to itself every cycle on a match.

For all the 351,023,283 non-exact matching reads, the SillaX traceback machine machine alignment results concur with the BWA-MEM's alignments with negligible (0.0023%) variance. On investigating the different alignments further, we noticed that the alignment scores produced by the SillaX traceback machine are exactly the same as that of BWA-MEM, implying that both alignments have the same mapping quality and should be treated the same. These differences are due to the fact that BWA-MEM and SillaX use different traceback techniques and policies for breaking ties when merging multiple paths with the same score.

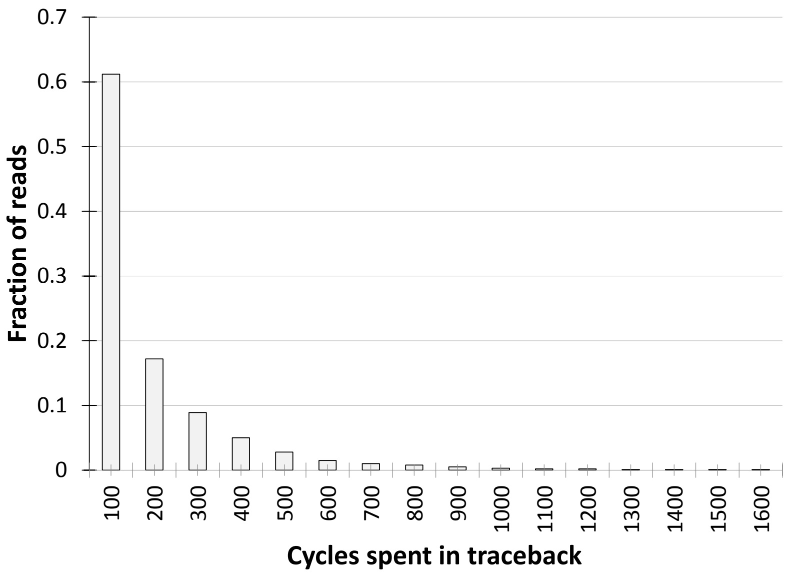


Figure 5.30: Silla traceback cycle distribution.

Broken pointer trail events: An important parameter to be considered while estimating the performance of the SillaX traceback machine is the number of times the machine must be re-executed because of a broken pointer trail. Across all the reads that we tested, we observe that only 7.59% of the reads require re-execution . This is consistent with our expectation. Figure 5.30 shows the distribution of cycles spent in re-execution. We can see that over 60% of the re-execution events are resolved within the first N (101) cycles. Thus, re-execution events have only a small impact on the performance of the SillaX traceback machine.

Throughput: Figure 5.31 shows the raw alignment throughput of the SillaX accelerator (4 lanes) when compared to banded Smith-Waterman based approximate string matching using SeqAn (CPU - 28 cores) and SW# (GPU - 3840 CUDA cores) when aligning 100bp Illumina short reads. It can be seen that SillaX achieves ∼62.9× throughput improvement over SeqAn and ∼5287× speedup over SW#. SillaX provides these speedups while consuming only 6.6 W of power and 5.64 mm^2 area. These benefits are both due to linear time processing of input symbols as well as efficient support for traceback. GPU-based solutions face high synchronization overheads for short reads leading to low performance.

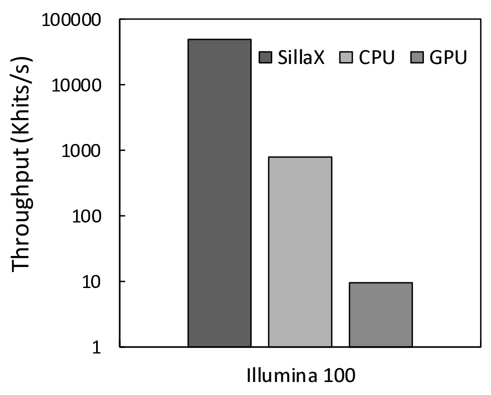


Figure 5.31: SillaX throughput (in Khits/s).

5.3 End-to-End Read Alignment Acceleration

In this section, we describe the improvements seen in end-to-end read alignment performance when combining our previous approaches to accelerate both seeding (set intersection, ERT) and seed-extension (Silla).

5.3.1 GenAx

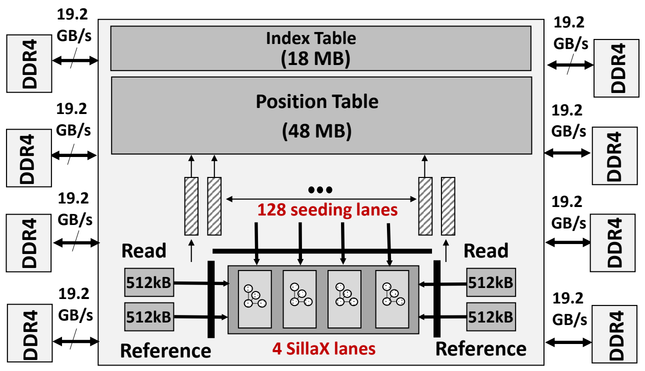


Figure 5.32: GenAx architecture overview.

GenAx brings together the set intersection-based seeding accelerator and SillaX seed-extension accelerators to enable high-throughput sequence alignment of human genomes. Figure 5.32 depicts the overall GenAx architecture. It consists of 128 seeding lanes which fetch k-mer positions from a 48 MB index table and k-mer reference hits from a 18 MB position table. Each indexing lane processes one read at a time. It has a CAM and a small control FSM for orchestrating SMEM intersections and k-mer lookups. The resulting hits after SMEM calculations are buffered for seed-extension by the SillaX lanes. GenAx features four SillaX lanes, which have sufficient throughput to process hits from all 128 seeding lanes. A SillaX lane fetches the reference string from the reference cache (4×512 KB) to extend a seed at a specific hit position. A 16 KB buffer (not shown) is used to buffer the reads processed.

The reference genome with 3 billion base-pairs is segmented into 512 segments. Therefore, each segment has 6 million base-pairs and a footprint of 1.5 MB which fits in the reference cache. Segments are processed sequentially for all reads. Before a segment starts, the position table, index table and reference for that segment are streamed in from memory via the 8 DDR4 channels shown. Since these are all spatially co-located memory accesses, streaming them in is efficient.

Throughput, Power and Area: Figure 5.33 (a) compares the overall throughput (reads/s) of GenAx with BWA-MEM (CPU) and CUSHAW2-GPU. It can be seen that GenAx achieves 31.7× speedup over BWA-MEM and 72.4× speedup over CUSHAW2-GPU. The large performance gains can be attributed to the following factors. (1) Efficient and composable SillaX accelerators accelerates seed-extension with in-place traceback. (2) Segmenting of index and position tables and storing them on-chip enables low-latency access and high-reuse across reads. (3) Read loading time takes a small fraction of the overall execution time (∼10%), increasing the benefits from segmenting. (4) Optimizing for the common case of perfect matches helps increase throughput.

Figure 5.33 (b) compares the average power consumption of GenAx vs. BWA-MEM and CUSHAW2-GPU. By sharing several indexing lanes with SillaX accelerators, GenAx reduces power consumption by 12× when compared to BWA-MEM execution on CPU.

Table 5.6 shows the area breakdown of GenAx. A large fraction of the die area is devoted for accelerating the seeding step using on-chip index and position tables. Each seeding lane consists of a 512-entry CAM. The SillaX lanes consist of a few counters and logic gates as described in Section 5.2.5.C. Overall, the GenAx architecture takes up 172.78 mm^2 in a 28nm node.

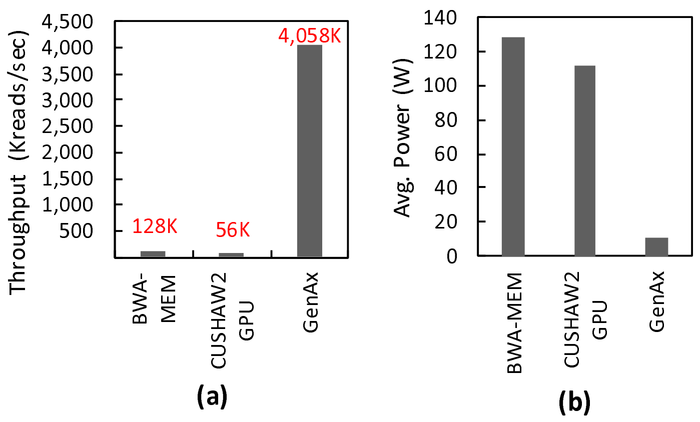


Figure 5.33: (a) Throughput comparison (in KReads/s) and (b) Power comparison.

Component	Area (in mm^2)
Seeding lanes (x128)	4.224
SillaX lanes (x4)	5.36
On-chip SRAM (68 MB)	163.2
Total	172.78

Table 5.6: Area breakdown: GenAx.

5.3.2 ERT + SeedEx [74]

While our ERT seeding accelerator can also be integrated with SillaX, in this section, we discuss the potential performance benefits obtained by integrating the ERT seeding accelerator with the area-efficient seed-extension accelerator (SeedEx) [74]. SeedEx improves upon SillaX since it only requires $\mathcal{O}(K)$ processing elements.

System	Instance	Throughput (Mreads/s)
BWA-MEM	c5n.18xlarge	0.216
BWA-MEM2	c5n.18xlarge	0.43
FPGA-ERT (best.) + SeedEx [74]	f1.4xlarge	0.903

Table 5.7: Overall read alignment performance on AWS EC2.

To match the seeding throughput of our FPGA implementation of ERT (FPGA-ERT), we augment ERT with 8 seed-extension accelerator lanes from SeedEx. Each seed-extension accelerator lane consists of 3 banded Smith-Waterman units (each with 41 PEs, `band-size=41`) and 1 edit-distance unit. Table 5.7 compares the overall read alignment performance of the software versions of BWA-MEM, BWA-MEM2 and our FPGA accelerated read alignment system. When integrated into BWA-MEM2, our FPGA accelerated read-alignment system provides 2.1× higher throughput compared to the software version of BWA-MEM2. Table 5.8 shows the resource consumption of both the seeding and seed-extension accelerators on one FPGA.

Component	LUT (%)	BRAM (%)	URAM (%)
Seeding Accelerator Total	36.71	26.42	61.66
Seed-Extension Accelerator Total	17.32	2.38	0.68
AWS Shell	19.74	12.63	12.20
Total	73.77	41.43	74.54

Table 5.8: FPGA-ERT + SeedEx: estimated resource utilization.
Estimated Per-FPGA Resource Utilization for FPGA-ERT + SeedEx [74].

System	Area Efficiency (KReads/s/mm^2)	Energy Efficiency (Reads/mJ)
BWA-MEM (CPU)	0.38	2.89
BWA-MEM2 (CPU)	1.13	8.59
CPU-ERT (best.)	2.32	17.56
ASIC-GenAx [73]	24.23	379.16
ASIC-ERT (best.)	276.36	347.51

Table 5.9: Seeding – area and energy efficiency comparison.

Table 5.9 compares the area efficiency and energy efficiency of CPU-BWA-MEM, CPU-BWA-MEM2, ASIC-GenAx [73] and the best configuration for ASIC-ERT. ASIC-GenAx is a recent sequence alignment accelerator that leverages CAM-based intersections to perform seeding. When compared to ASIC-GenAx [73] which use large on-chip SRAMs, ASIC-ERT uses lightweight tree walker units and improves area efficiency by 11.4×.

5.4 Summary

Genomics is at an inflection point. Over the next decade, it is conceivable that every individual's genome would be sequenced and analyzed. Given that a single human genome generates 100 GB – 1TB of data, we need orders of magnitude improvement in computing efficiency to analyze this data and realize the full potential of genomics. This work takes an important step towards this goal by accelerating read alignment. Our read alignment accelerator GenAx provides a throughput of 4,058K reads/s for Illumina 101 bp reads. GenAx achieves 31.7× speedup over the standard BWA-MEM sequence aligner running on a 56-thread dual socket 14-core Xeon E5 server processor, while reducing power consumption by 12× and area by 5.6×. Our ERT-based seeding accelerator implemented on AWS F1 cloud when combined with SeedEx seed-extension accelerators can achieve 2.1$\times$ speedup over state-of-the-art read alignment software BWA-MEM2 while maintaining binary compatibility. We open source the ERT software implementation and integrate it into BWA-MEM2 (ert branch: https://github.com/bwa-mem2/bwa-mem2/tree/ert).

CHAPTER 6

GenomicsBench: Characterizing the Genomics Computing Landscape

Previous chapters discussed acceleration approaches for two of the key computationally intensive pattern matching kernels used in genomics analysis pipelines: (1) seeding: an exact matching kernel and (2) seed-extension: an approximate string matching kernel. Apart from these two, there are several other computationally intensive genomics kernels that can also benefit from hardware acceleration. In this work, we identify such kernels and perform a detailed architectural characterization to identify performance bottlenecks. We also provide an open-source implementation of these kernels along with representative inputs as a benchmark suite to facilitate future algorithm optimizations and architectural exploration.

6.1 Need For A Genomics Benchmark Suite

This work aims to identify commonly used modern sequencing pipelines, characterize their performance and extract their compute-intensive kernels. The goal is to compile a standardized genome sequencing benchmark suite that highlights the growing compute need in genomics and helps shape future computing research in this space. Such an effort has been lacking for this important computational domain. Some notable prior works that perform detailed architecture characterization of important bioinformatics workloads such as BioPerf [40], BioBench [37] and MineBench [138] were carried out in the last decade when sequencing technologies were still nascent and not so diverse. Modern sequencing pipelines have vastly different bandwidths, latencies, portability requirements, algorithms, and pipelines than those used a decade ago. For instance, new kernels that leverage vectorized implementations for dynamic programming are now common. Machine learning algorithms are now widely used to process long but noisy reads. There is a wide variety of sequencers that vary in terms of throughput, read length, and accuracy, to meet different medical research and clinical needs. These have resulted in a plethora of bioinformatics tools and pipelines. Without a standardized benchmark suite that represents common computational kernels, it becomes increasingly difficult to design efficient computing system and processor architectures for this rapidly emerging domain. There is also growing interest in developing custom hardware solutions for sequencing [21]. These efforts can also greatly benefit from the availability of a genomics benchmark suite.

6.2 GenomicsBench Benchmark Suite

FM-Index Search (fmi): The FM-index (Full-Text Index in Minute Space) is one of most common data structures in aligners such as Bowtie2 [112], BWA-MEM [118, 128], SOAP3-dp [125] and metagenomics classification tools such as Centrifuge [100]. It is used to identify the locations of short matching substrings of the read (called *seeds*) in the reference genome. The FM-index is attractive because of its low memory footprint, ability to match substrings of any length and support for inexact matching (i.e., identifying seeds with a small number of edits with respect to the reference).

Figure 6.1 (a) shows the FM-index constructed for a sample reference (R) and an example search query from the read. The FM-index consists of: (1) the *suffix array (SA)*, which contains the locations of lexicographically sorted suffixes of the reference genome `R`, (2) the *Burrows Wheeler Transform (BWT)*, computed as the last column of the sorted suffix array of the reference, (3) the *count table (C)* which stores the number of characters in `R` lexicographically smaller than a given character `c` and (4) the *occurrence table (Occ)* which stores the number of occurrences of a

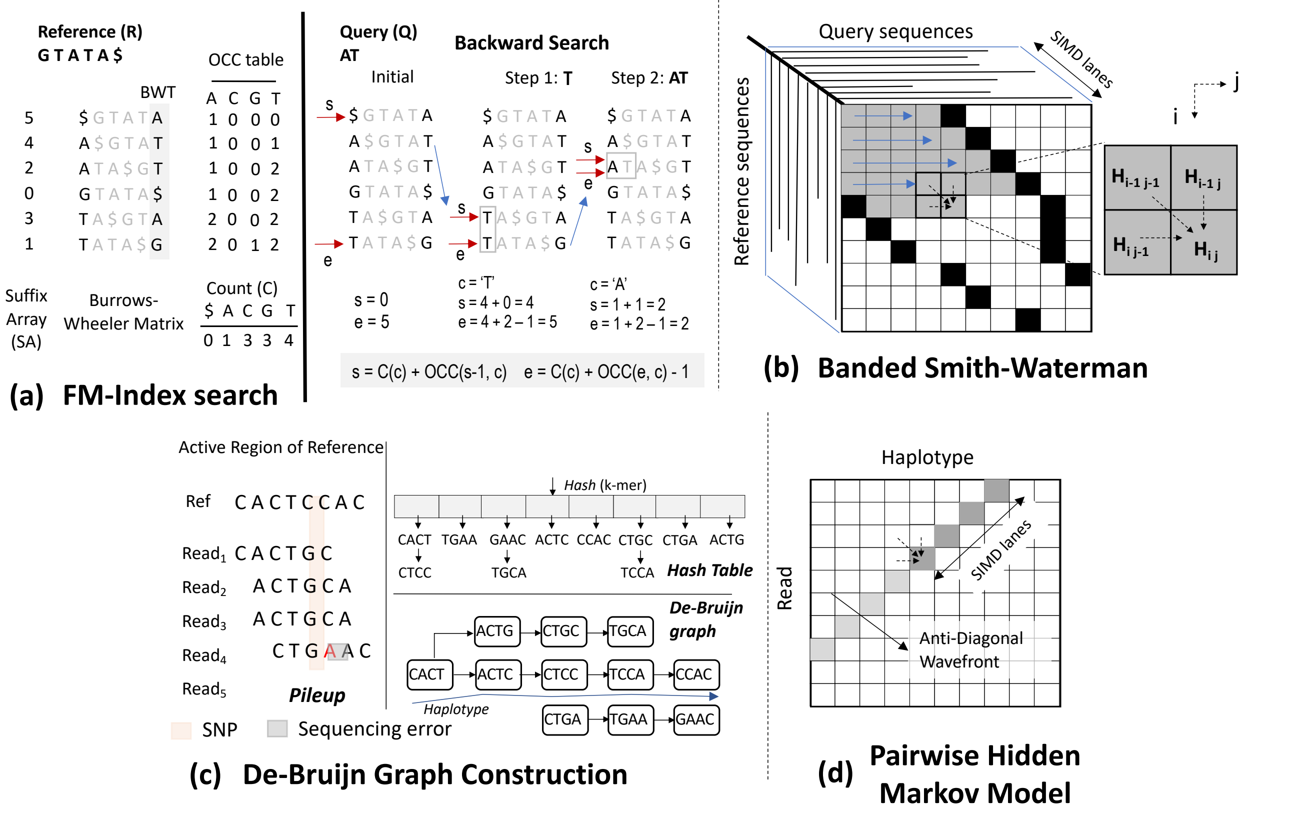


Figure 6.1: Benchmark kernels in GenomicsBench (1)
(a) FM-index search. (b) Banded Smith-Waterman. (c) De-Bruijn Graph construction. (d) Pairwise Hidden Markov Model.

character up to a certain index in the *BWT* array.

The FM-index allows the backward search of a query of length ($|Q|$) in $\mathcal{O}|Q|$) iterations, with at most 2 memory lookups per iteration (one each for computing the start and end (s, e) intervals of the match). It is characterized by irregular memory accesses to the large Occ table (blue arrows in Figure 6.1 (a).) and is both memory-latency and memory-bandwidth bound. Since the memory access characteristics of FM-index search are similar across different tools, we choose the optimized super-maximal exact match (SMEM) search computation in BWA-MEM2 [128] in our benchmark suite. SMEM computation uses the FM-index to find the longest exact match spanning a given position in the read.

Input Datasets: We provide small and large datasets, which are a set of 1M and 10M human reads respectively, each 151 bases long, from sample SRR7733443 [128].

Banded Smith-Waterman (bsw): The Smith-Waterman algorithm [154] is a dynamic-programming algorithm that estimates the pairwise similarity between pairs of sequences X and Y with lengths m and n respectively in $\mathcal{O}(mn)$ time and space. It is commonly used in sequence alignment tools like BWA-MEM [118] and variant calling tools like GATK Haplotype Caller [8, 9] to align millions to billions of sequence pairs and is a major computation bottleneck. The similarity score for DNA sequences is typically computed using affine gap penalties [134], which uses different penalties for different edits (i.e., substitution, insertion and deletion) and allows for identification of biologically meaningful short insertions/deletions in pairwise alignments. It requires the computation of three matrices H, E and F corresponding to the different edit types. For aligning sequences with a maximum of w insertions/deletions, a banded version of the Smith-Waterman algorithm is commonly used (Figure 6.1 (b). region between the black squares) reducing time and space complexity to $\mathcal{O}(wn)$ where w is the width of the band of cells computed in each row.

$$\begin{aligned}
H_{ij} &= \max\{H_{i-1,j-1} + s(i,j), E_{ij}, F_{ij}\} \\
E_{i+1,j} &= \max\{H_{ij} - q, E_{ij}\} - e \qquad . \\
F_{i,j+1} &= \max\{H_{ij} - q, F_{ij}\} - e
\end{aligned} \tag{6.1}$$

Equation 6.1 shows the recurrence relation for the Smith-Waterman algorithm. $s(i, j)$ is a pre-computed similarity score between characters $X[i]$ and $Y[j]$ and the score in cell (i, j) of matrix H (i.e., H_{ij}) is the similarity score for substrings $X[0, i]$ and $Y[0, j]$. We choose the optimized banded Smith-Waterman implementation in BWA-MEM2 [128] for our benchmark suite. It makes use of inter-task parallelism to allocate similarly sized sequence pair tasks to different SIMD lanes.
Input Datasets: Our small and large datasets use 100K and 10M seed-extension pairs obtained from inputs to the Smith-Waterman function in BWA-MEM2 for reads from the human sample SRR7733443 [128].

K-mer Counting (kmer-cnt): A k-mer is a fixed k-length substring of a DNA sequence. K-mer counting counts the number of occurrences of each unique k-mer in the input reads. It is one of the most common tasks in bioinformatics sequence analysis and is widely used in de novo assembly [105, 107], error correction [129] and metagenomics classification [141]. Common use cases include filtering out low-frequency k-mers in the input data that are likely to be sequencing errors, finding high-frequency k-mers characteristic of repetitive genomic regions and constructing k-mer histograms to serve as signatures of the input data [180]. Typical k-mer lengths are 15-55. The computation task in k-mer counting is an incremental update to a hash-table for each k-mer. These updates can be parallelized across millions to billions of k-mers in the input dataset. We focus on shared-memory k-mer counting and characterize the k-mer counting implementation in the popular Flye assembler [105].

Input Datasets: Our small and large datasets use 1K and 50K Oxford Nanopore reads from *E.coli* sequenced by Loman lab [7].

De-Bruijn Graph Construction (dbg): Prior to calling variants using the reads aligned to a region of the reference genome (e.g., $\sim$100-1000 bases), it is necessary to correct read alignment artifacts. Modern variant callers like GATK Haplotype Caller [8, 9] and Platypus [147] do this by reassembling those reads into a De-Bruijn graph and later traversing this graph to generate strings that are likely to contain variants (called *haplotypes*). The graph is constructed from both the k-mers of the read and the reference as shown in Figure 6.1 (c). Each node in the graph represents a unique k-mer and each edge links adjacent k-mers in the input. A hash table is used to track nodes that have already been inserted into the graph. If cycles are found in the graph, graph construction is repeated by increasing the k-mer size. Each input task to this kernel is a set of reads aligned to a reference region. The reassembly tasks can be parallelized across different regions. We model the De-Bruijn graph construction implementation in the Platypus variant caller [147] for the benchmark suite that accounts for >60% of its execution time.

Input Datasets: We use BWA-MEM aligned records from the Platinum Genomes dataset [67]. Our small dataset uses a region of chromosome 22 (bases 16M-16.5M) while the large dataset uses the entire chromosome 22.

Pairwise Hidden Markov Model (pairHMM): Using the reads aligned to a region of the reference genome and the candidate haplotypes identified from De-Bruijn graph traversal, a pairwise alignment of each read to each candidate haplotype is performed to identify the most likely haplotypes supported by the reads. The total workload per region is $|R| \times |H|$ pairwise alignments, where $|R|$ and $|H|$ are the number of reads and haplotypes respectively. Pairwise alignment is performed using a Hidden Markov Model (HMM) and the likelihood score is computed using the following dynamic-programming recurrence relations [143]:

$$\begin{aligned} M_{ij} &= (M_{i-1j-1}\theta \ + I_{i-1j-1}\kappa \ + D_{i-1j-1}\lambda) \cdot P_{ij} \\ I_{ij} &= M_{i-1j}\tau \ + I_{i-1j}\epsilon \\ D_{ij} &= M_{ij-1}\zeta \ + D_{ij-1}\eta \end{aligned} \quad . \tag{6.2}$$

where: M_{ij}, I_{ij} and D_{ij} represent match, insertion and deletion probabilities for aligning read substring $R[0,i]$ to haplotype substring $H[0,j]$, where $0 \leq i \leq |R|$ and $0 \leq j \leq |H|$. These are weighted by different transition and emission parameters of the HMM: $\theta, \kappa, \lambda, \tau, \epsilon, \zeta, \eta$. P_{ij} is the prior probability of emitting bases $(R[i], H[j])$, computed using the floating point base-quality scores for the read R. Base-quality scores are typically provided by the basecaller and indicate the confidence of the basecaller in calling each base in the read. Low quality bases from the read contribute a smaller amount to likelihood score computed above. The computation in pairHMM differs from the Smith-Waterman kernel described earlier mainly in the use of floating-point computation. There exists abundant intra- and inter-task parallelism in this workload. Intra-task parallelism arises from data-parallel processing of cells along the wavefront as shown in Figure 6.1 (d). Inter-task parallelism arises by parallel processing of different genome regions. We use the optimized SIMD implementation in GATK Haplotype Caller [8] as part of the benchmark suite and extend it to leverage inter-task parallelism using multiple CPU threads.

Input Datasets: We use the read-haplotype pair inputs to the *calcLikelihoodScore* function in GATK Haplotype Caller [8]. Our small dataset uses as input BWA-MEM aligned reads for region chromosome 22:16M-16.5M, while the large dataset uses reads aligned to the entire chromosome 22.

Chaining (chain): One of the most time-consuming steps in de novo assembly of long reads is overlap estimation between reads [105, 107]. We characterize the chaining implementation from

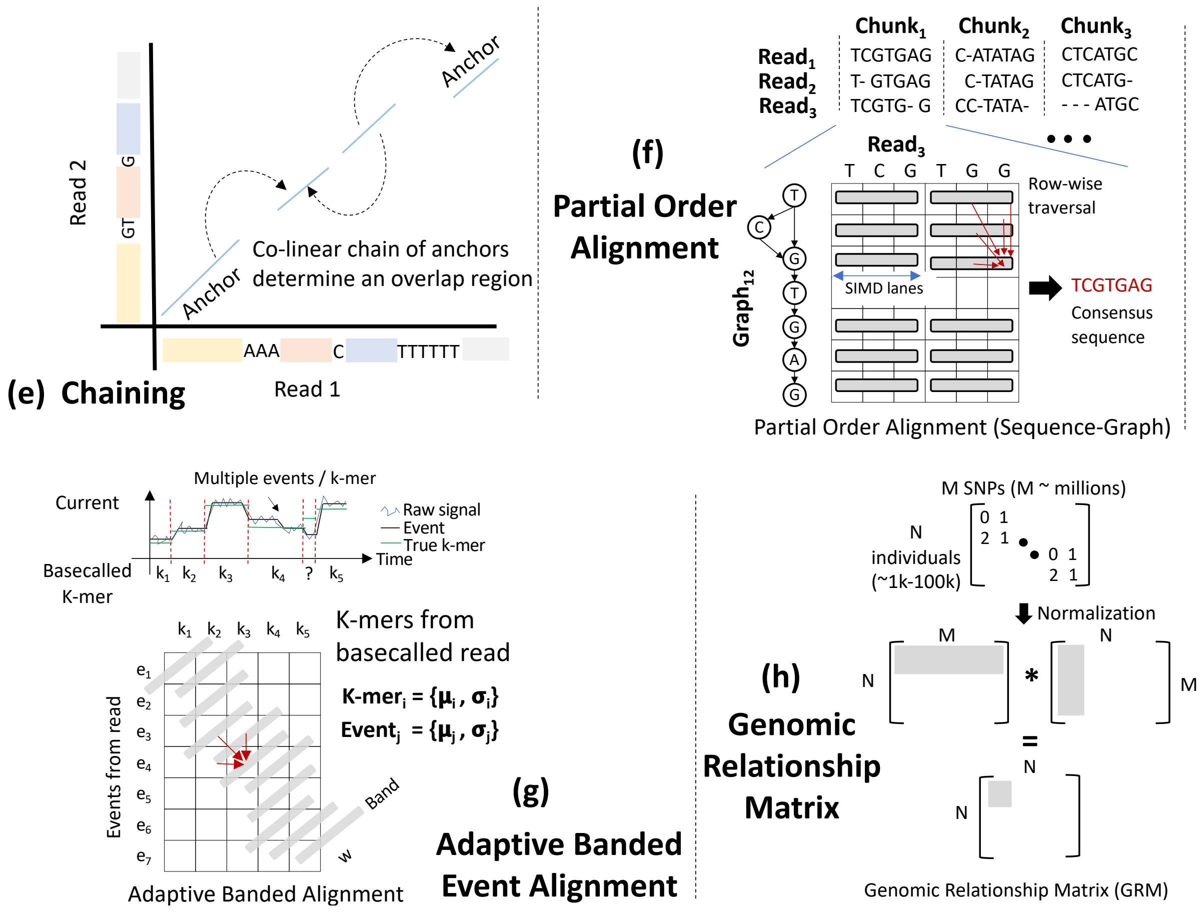


Figure 6.2: Benchmark kernels in GenomicsBench (2).
(e) Chaining. (f) Partial Order Alignment. (g) Adaptive Banded Event Alignment. (h) Genomic Relationship Matrix.

Minimap2 [119] which is one of the most popular tools for estimating pairwise overlap between reads and extend it to support inter-task parallelism across different pairs of reads. Given a set of seeds (also called *anchors*) shared between a pair of reads, chaining aims to group together a set of co-linear seeds into a single overlapping region as shown in Figure 6.2 (e). The chaining algorithm is a 1D dynamic programming based algorithm that compares each anchor with N previous anchors (default = 25) to determine its best parent. The recurrence relation used to estimate the maximal chaining score of the i^{th} anchor [119, 79] is:

$$score(i) = \max\left\{ \max_{i>j\geq 1}\{score(j) + \alpha(j,i) - \beta(j,i)\}, w_i\right\} \tag{6.3}$$

where w_i is the length of anchor i, $\alpha(j,i)$ is the number of matching bases between anchors i and j

after accounting for overlaps between them and $\beta(j, i)$ is a penalty that is set based on the relative distance between a pair of anchors on the two reads.

Input Datasets: Our input dataset uses the anchors for 1K and 10K reads from the PacBio sequence data for the *C.elegans* worm [6, 79] when computing overlaps with itself.

Partial-Order Alignment (poa): After assembling the reference genome of a new species, it is common to perform a *polishing* step to correct small errors in assembly using the aligned reads. Racon [164] is one of the most popular tools for long-read polishing. Given a set of reads aligned to the target genome, Racon first splits the reads into non-overlapping windows called chunks (which can be processed in parallel) and then incrementally constructs a partial-order graph [114] by aligning new sequences to it using a SIMD accelerated dynamic programming algorithm (see Figure 6.2 (f)). Later, the consensus sequence is generated from the graph using the heaviest bundle algorithm [113]. Each node in the partial-order graph represents a base of the input sequence and weighted edges represent support from different reads in the chunk. Since the nodes in multiple branches of the graph cannot be ordered relative to each other, the graph is said to be *partially ordered*. Aligning new sequences to the graph is the most time-consuming operation in Racon and has complexity $\mathcal{O}((2n_p + 1)n|V|)$, where n_p is the average number of incoming edges to nodes in the graph, $|V|$ is the number of nodes in the graph and n is the length of the read chunk. Contrast this with Smith-Waterman which has complexity $\mathcal{O}(mn)$, with regular data-dependencies. As used in Racon, our `poa` benchmark builds the consensus sequence for each chunk in a separate CPU thread.

Input Datasets: We use 1000 and 6000 consensus tasks for our small and large datasets respectively. These are obtained when polishing the Flye-assembled *Staphylococcus aureus* genome with Minimap2-aligned ONT long reads [175].

Adaptive Banded Signal to Event Alignment (abea): Comparing a time-series of raw nanopore signal data to a reference genome sequence is a common task in the polishing of long-read sequencing data and the detection of methylated bases (i.e., non-standard nucleotides apart from A, C, G, T, which play an important role in controlling gene expression). After segmenting the signal data into different *events* based on sudden changes in signal current, each event is then compared against the k-mers of the reference genome using a computationally intensive dynamic programming algorithm called *adaptive banded event alignment (ABEA)* [75]. ABEA is the most time-consuming kernel when performing methylation calling using the software tool Nanopolish [122]. Event alignment is more complex than banded sequence alignment since it requires an adaptive band [158] to capture long gaps in optimal alignments especially when dealing with long and error-prone Nanopore reads. These long gaps arise because k-mers are often over-represented (up to $2\times$) by multiple events as they are sampled by the nanopore. Furthermore, event alignment uses 32-bit floating point log-likelihood computation in its scoring function and is computationally more expensive than sequence alignment. We analyze the optimized GPU implementation of ABEA [75] as part of the benchmark suite. In this heavily optimized implementation, ABEA accounts for 24.5% of total runtime.

Input Datasets: For ABEA, our small and large datasets use 1,000 and 10,000 raw FAST5 reads from chromosome 22 of NA12878, and the GRCh38 reference genome. This data was obtained from the publicly available "Nanopore WGS Consortium" dataset [23, 91].

Genomic Relationship Matrix (grm): All large-scale population genomics studies need to account for potential ancestral relationship between individuals in the study. This is done by computing a $N \times N$ matrix called Genomic Relationship Matrix (or GRM), where N in the number of individuals in the study. Each element of the GRM G_{ij} describes the average genetic similarity between individuals and is computed as follows:

$$G_{ij} = \frac{1}{S} \cdot \sum_{s=1}^{S} \frac{(x_{is} - 2p_s)(x_{js} - 2p_s)}{2p_s(1 - p_s)} \tag{6.4}$$

where x_{is} and x_{js} indicate the number of copies of the non-reference base at location s for individuals i and j respectively and p_s is expected frequency of a non-reference base at location s in the population. S is the total number of SNV (Single Nucleotide Variation) location markers in the reference genome. We extract the GRM kernel from the popular population genomics software PLINK2 [51]. The kernel performs dense matrix multiplication and can benefit from parallel computation of different output elements as shown in Figure 6.2 (h).

Input Datasets: We compute the GRM on SNV data belonging to 2504 individuals from 1000 Genomes Project Phase 3 [51]. Our small dataset uses 194K variants from chromosome 22 and our large dataset uses 1.07M variants from chromosome 1.

Neural Network-based Base Calling (nn-base): When performing nanopore based genome sequencing, raw nanopore signal data must be correctly converted to a sequence of nucleotide bases through a process called *basecalling* discussed earlier. As DNA moves through a nanopore, it does so at a highly variable rate, and the resulting current is affected by multiple consecutive nucleotides occupying the pore ($\sim$5-10, depending on the pore chemistry). Due to the limited resolution of the Analog-to-Digital Converter (ADC) sampler and unavoidable background noise, there is considerable overlap between current levels measured for different 5-mers. Basecallers resolve this ambiguity in two stages. First, a deep recurrent or convolutional neural network aggregates contextual information to determine the most likely nucleotide observed at each time step. Using these probabilities, a connectionist temporal classification decoder [78] then determines the most likely sequence. The neural network is by far the most time-consuming basecalling stage. In order to make this computation regular and parallelizable, existing basecallers segment the signal and perform inference on many independent chunks, stitching the final sequence together as a post-processing step. Our benchmark includes the GPU-based CNN basecaller Bonito, which currently boasts the highest basecalling accuracy [4].

Input Datasets: For basecalling, our small and large input datasets are 100 and 1,000 raw FAST5 reads from chromosome 20 of NA12878. This data was obtained from the publicly available "Nanopore WGS Consortium" dataset [23].

Pileup Counting (pileup): A common pre-processing step in long-read neural network variant callers such as Medaka [14] involves parsing of alignment data for all reads aligned to a region of the reference genome (called *read pileup*) and generating counts for different bases, insertions and deletions at these different pileup locations. These counts are later analyzed by the recurrent

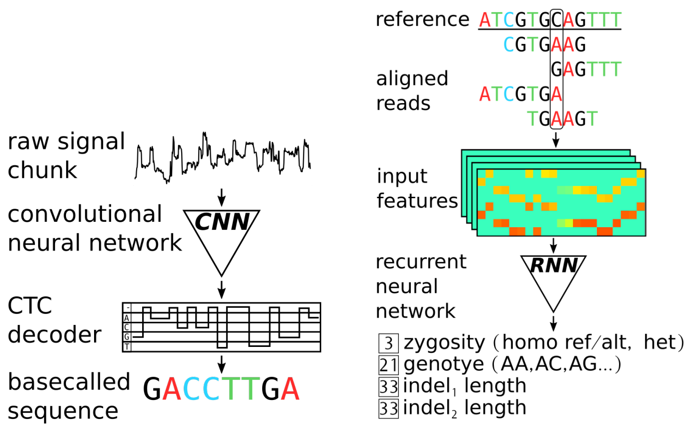


Figure 6.3: Neural Network-based algorithms in Bonito and Clair. Overview of Bonito (left) and Clair (right).

neural network to call variants. This pre-processing step is time-consuming because it involves random access into the alignment record to extract and parse alignment information (represented as a CIGAR string [32]). Fortunately, the pre-processing step can leverage inter-task parallelism by distributing the processing of different 100 kilobase regions of the reference genome to different CPU threads. The benchmark suite includes the inter-task parallel version of pileup counting.

Input Datasets: We use the results from Minimap2 alignment of ONT reads. Our small dataset uses aligned reads to the *Staphylococcus aureus* genome [175], while the large dataset uses reads aligned to chromosome 20 of sample HG002 [28].

Neural Network-based Variant Calling (nn-variant): Long-read variant callers examine the read pileup for a particular genome reference position and call variants with respect to that reference. We chose to analyze the Clair variant caller because it outperforms competing tools in terms of both performance and accuracy for long reads [124]. As input, Clair accepts a size $33 \times 8 \times 4$ tensor. Given a particular reference position, this tensor is generated using pileup information for 16 bases flanking each side ($16 + 1 + 16 = 33$), and considering the pileup counts for each base (A,C,G,T) and strand (forward, reverse) individually ($2 * 4 = 8$). Furthermore, 4 different encodings of the same information is used: **(a)** raw pileup counts, **(b)** support for insertions relative to (a), **(c)** support for deletions relative to (a), and **(d)** support for alternative variants relative to (a). Clair uses a series of recurrent neural networks with bidirectional long short-term memory

(LSTM) units and fully-connected layers to predict a potential variant. Refer to [124] for network details.

Input Datasets: For benchmarking Clair on long reads, we selected all raw FAST5 reads from the q13.12 region of chromosome 20 of NA12878 from the “Nanopore WGS Consortium” [23] dataset. These reads were basecalled using high-accuracy Guppy 3.6.0, and mapped using Minimap2. Our small dataset variant called the first 10,000 reference positions from this region, and our larger dataset used 500,000.

6.3 Performance Characterization of Benchmarks

6.3.1 Characterization Methodology

Several of the genomic analysis tools described earlier operate on large datasets and can run for several days. To keep the study manageable, we adopt the following methodology. We first profile all software tools with Intel VTune Profiler 2020 [13] as well as manual timing instrumentation to identify the most time-consuming kernels in both single and multi-thread settings. Later, we isolate these kernels and run representative input datasets of two sizes. Kernel executions with the `small` inputs finish in a few minutes, while the `large` inputs take 5–20 minutes on a single-thread. Both the `small` and `large` inputs capture the bottlenecks in the original application and exercise the kernel in similar ways (e.g., similar proportions of different dynamic instructions and memory accesses with different strides). We use the MICA pintool [89] to compute statistics on the instruction distribution in these benchmarks. Cache miss and memory stalls are obtained using performance counter events from the hardware event-based sampling collector [31]. All kernels and inputs/outputs are extracted as-is from the original software tools. The tools already support multithreading. For ease of benchmarking, we made the following modifications to the extracted benchmarks: (1) OpenMP parallelization with dynamic scheduling was used to reliably evaluate thread scaling of the benchmark after isolation from the software tool and (2) file I/O-related driver code was added for reading inputs and writing results. GPU benchmarks are characterized using Nvidia’s Visual

Profiler [26] and *nvprof* on the Nvidia Titan Xp GPU with 12 GB GDDR5x memory.

CPU	Intel Xeon E3-1240 v5 3.5 GHz; AVX2; 1 socket; 8 threads
L1 I&D cache	4 x 32KB Inst; 4 x 32KB Data, 8-way
L2 cache	4 x 256KB, 4-way
L3 cache	4 x 2 MB, 16-way
Memory bandwidth	31.79 GB/s

Table 6.1: Baseline system configuration used for characterization.

Table 6.1 details our experimental machine configuration. We present characterization results for all benchmarks except `nn-variant` which failed to complete successfully using `nvprof` on both a native run as well as within a Docker container.

6.3.2 Parallelism

6.3.2.A CPU benchmarks

In this section, we present a detailed characterization of the sources of parallelism in our CPU benchmarks and the challenges in exploiting them.

Benchmark	**Input Datatype**	**Applications**	**Chosen Tool**	**% Time Spent in Tool** (single-thread)	**Parallelism Motif**
fmi	Short reads	Read Alignment, Metagenomics Classification	BWA-MEM2	38%	Tree Traversal
bsw	Short reads	Read Alignment, De-Novo Assembly	BWA-MEM2	31%	Dynamic Programming
dbg	Short reads	Variant Calling, De-Novo Assembly	Platypus	65%	Graph Construction, Hash Table
phmm	Short reads	Variant Calling, Error Correction	GATK Haplotype Caller	70%	Dynamic Programming
chain	Long reads	De-Novo Assembly, Read Alignment	Minimap2	47.4 %	Dynamic Programming (1D)
spoa	Long reads	Error Correction	Racon	75 %	Dynamic Programming, Graph Construction
abea	Long reads	Basecalling, Variant Calling	Nanopolish	71.4%	Dynamic Programming
grm	NA	Population Genomics	PLINK2	92.8 %	Dense Matrix Multiplication
nn-base	Long reads	Basecalling	Bonito	95 %	FP Matrix Multiplication
nn-variant	Long reads	Variant Calling	Clair	57.2 %	FP Matrix Multiplication
kmer-cnt	Long reads	De-Novo Assembly	Flye	10%	Hash Table
pileup	Long reads	Variant Calling	Medaka	6.3 % [1]	—

Table 6.2: GenomicsBench: Benchmark characteristics.
Categorization of benchmarks. For benchmarks with utility in more than one application, the selected application is underlined.

Table 6.2 presents an overview of the different benchmarks and their corresponding parallelism motifs based on the taxonomy provided in [180]. `bsw`, `phmm`, `chain`, `spoa` and `abea` are based on dynamic programming, but have important differences. The key differences are: (1) type of data dependency present (e.g., 1D / 2D), (2) amount of computation needed (e.g., banded / full matrix), (3) type of matrix traversal (e.g., wavefront / row-wise) and (4) type of input (e.g., sequence / graph). Also present in the benchmark suite are kernels that manipulate hash tables and perform graph construction (`dbg`, `spoa`).

Some of the GenomicsBench benchmarks like `grm` and `kmer-cnt` have **regular compute** patterns since their inputs come in regular, pre-determined sizes. In contrast, a majority of the GenomicsBench benchmarks work on inputs with varying sizes and characteristics and have **irregular compute** patterns.

Benchmark	Parallelism Granularity	Data-Parallel Computation
fmi	Read batch	# OCC Table Lookups
bsw	Seed	# Cell Updates
dbg	Genome Region	# Hash Table Lookups
phmm	Genome Region	# Cell Updates
chain	Read	# Input Anchors
spoa	Read Chunk Window	# Cell Updates
pileup	Genome Region	# Read Lookups

Table 6.3: Sources of parallelism for different benchmarks.
Parallelism granularity and data-parallel computation for irregular CPU benchmarks. Other regular compute benchmarks not shown.

Table 6.3 shows the data-parallellism granularity for each of the irregular compute GenomicsBench benchmarks and the corresponding data-parallel computation performed. Note that it is possible to reduce the data-parallelism granularity further by vectorizing each of the data-parallel computations shown in the second column of Table 6.3. However, this comes with significant additional complexity arising from the complex data dependencies present in the benchmarks (Figures 6.1 and 6.2). To overcome this, implementations often speculate on the absence of data dependencies to achieve high performance (e.g., [70]).

This complexity can often be traded off for abundant parallelism to be exploited across two other dimensions: (1) read-level parallelism and (2) genome region-level parallelism as shown in Table 6.3. Since each of the benchmarks process millions to billions of reads across millions of genome regions there exists abundant data-parallelism across both these dimensions. Several software tools have adopted this approach. For example, BWA-MEM2 [128] has demonstrated significant benefits by vectorizing inter-sequence computation instead of vectorizing the cell updates for `bsw`.

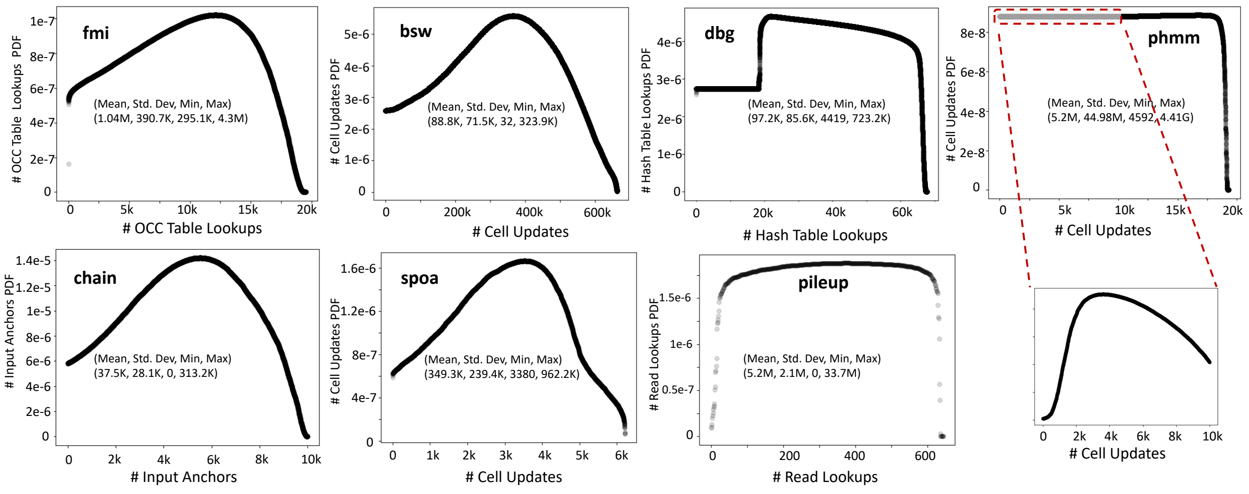


Figure 6.4: Variation in the amount of computation performed for each task.
Distribution of the amount of data-parallel computation performed for each task (x-axis) and its frequency (y-axis) for the different benchmarks. Variations in the computation needed for each task based on its size and input data characteristics makes it difficult to exploit the abundant parallelism present in each benchmark. To enable comparison across benchmarks, the normal probability distribution function (PDF) has been used to represent the frequency of computation on the y-axis.

Challenges in exploiting data parallelism: In spite of the abundant read-level / genome region-level parallelism in the GenomicsBench benchmarks, it is difficult to exploit them effectively in different software tools. To understand why, consider the following hypothetical scenario where each data-parallel computation entity discussed in Table 6.3 is assigned to a separate vector lane. Each vector is replaced with a new batch of tasks as soon as all of the ones currently assigned to it complete. For vectorization to be efficient, all the tasks assigned to each lane must perform a similar amount of computation. Any imbalances in the computation across vector lanes can severely reduce the efficiency of vector computation and lead to control divergence. For this reason, the inputs to the `bsw` kernel, for example, are sorted based on sequence lengths before being assigned to SIMD lanes. However, even if differences in input sequence lengths have been accounted for, differences in input sequence content can greatly influence the computation performed in each SIMD lane. This is because the matrix computation can also be aborted early when aligning highly dissimilar sequences of similar length using `bsw`. As a result we find that the AVX2 16-bit inter-sequence vectorized `bsw` implementation in GenomicsBench performs 2.2× more cell updates than the scalar implementation. Note that the vectorization challenges outlined above exist not only for CPU-based software tools but also for GPUs, which also employ SIMD units to increase compute density.

Similar observations can also be made for the other irregular CPU compute benchmarks. It can be seen from Figure 6.4 that there exists significant variation in the amount of data-parallel computation performed by different tasks in different benchmarks. For `phmm`, which computes the most likely haplotype given supporting reads, certain genome regions can have up to 1000× imbalance in the computation needed when compared to the average case (as can be seen from the mean ($5.2M$) and maximum ($4.41G$) cell update values across different regions). However, it must be noted that regions with such low or high computational demand are fewer (as indicated by the lightly shaded circles). Across different benchmarks we find that the ratios of maximum to average computation per task can vary from 4.1× to 8.3×.

6.3.2.B GPU benchmarks

Whereas the CPU kernels selected for this benchmark suite were diverse and often encountered challenges in exploiting data parallelism, the GPU kernels we investigated had fairly regular control flow and compute patterns. Predictable control flow and data accesses are a prerequisite for efficient utilization of GPU computing resources, and `abea` and `nn-base` were likely implemented on the GPU for this reason.

	abea	**nn-base**
Branch efficiency	100 %	100 %
Warp efficiency	75.09 %	100 %
Non-predicated warp efficiency	70.18 %	94.43 %
SM utilization	70.53 %	99.83 %
Occupancy	31.41 %	88.47 %

Table 6.4: GPU kernel control flow and compute regularity.

The `abea` and `nn-base` kernels both avoid branch divergence entirely, and achieve relatively high warp efficiency. This is shown in Table 6.4. Warp efficiency is defined as the average fraction of active threads in a warp, and "non-predicated" efficiency restricts the definition of active to threads which are not executing predicated instructions. Neural network basecallers such as Bonito break sequences of raw nanopore signal into regular chunks of 4,000 consecutive measurements and feed that data into a fixed-size neural network. Since floating point matrix multiplication is computationally intensive and involves very little control flow, `nn-base` is able to achieve perfect warp efficiency and nearly-complete occupancy and SM utilization. The few predicated instructions reducing overall throughput are likely due to the fact that the neural network of `nn-base` does not operate using filters of sizes which are integer multiples of 32, the number of threads in a warp. On the other hand, the `abea` kernel performs a dynamic programming matrix computation instead of matrix multiplication, it is limited by the execution and memory dependencies inherent to the structure of the computation. Furthermore, `abea` requires frequent synchronization between warps. As a result, the SM utilization and warp efficiency are lower.

6.3.3 Instruction Diversity

Instruction diversity characterization helps determine the complexity of functional units needed for specialized hardware. Figure 6.5 shows the dynamic instruction breakdown for the different CPU benchmarks. The "Other" category includes string, system call, prefetching, and synchronization instructions.

Among the benchmarks analyzed, `phmm`, `bsw`, and `spoa` benefit from SIMD vectorization and have a high proportion of vector computation instructions. It can be also be seen that `phmm` is the only CPU kernel that performs floating point computation, while the other kernels are dominated by scalar integer computation. `phmm` also uses single-precision floating point computation in most cases, and resorts to double-precision floating point only in rare cases when single-precision is insufficient to represent the result. `bsw`, `phmm`, and `chain` are compute-intensive and have a lower proportion of memory loads and stores when compared to memory-intensive benchmarks like `fmi`. We also looked at the common operations performed in vectorized benchmarks. For instance, `bsw`

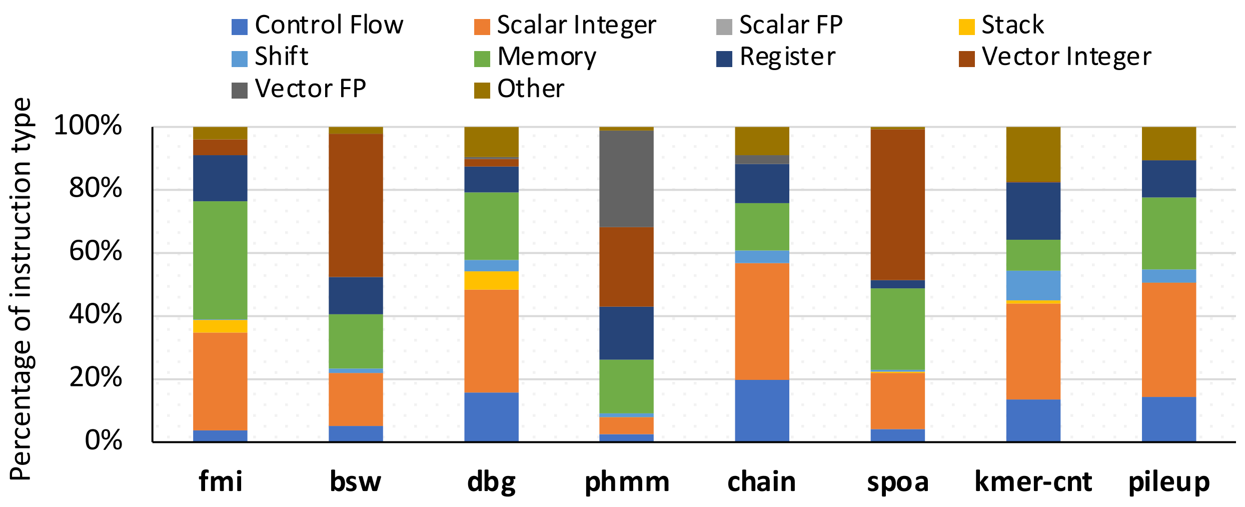


Figure 6.5: Dynamic instruction breakdown for different benchmarks.
Breakdown of dynamic instructions in different benchmarks. `grm` is excluded because its multi-threaded design to decompress inputs affects the accuracy of measurements from the MICA pintool.

uses `blend` instructions for cell updates and band adjustment, and `spoa` extensively uses `shift` instructions to compare against cells present in a previous column or diagonal but which are part of a different SIMD vector.

6.3.4 Memory Access Characteristics

6.3.4.A CPU benchmarks

In this section, we perform a detailed characterization of the memory access patterns of different GenomicsBench benchmarks.

Off-chip Data Requirements: Figure 6.6 shows the off-chip data requirements for different GenomicsBench benchmarks. It can be seen that benchmarks like `fmi` and `kmer-cnt` have significantly higher off-chip data requirements, measured in DRAM bytes per kilo-instruction (BPKI) (66.8 BPKI and 484.1 BPKI respectively). For `fmi` and `kmer-cnt` the memory access bottlenecks are due to irregular memory accesses over large working sets, ∼10 GB (FMD-index in BWA-MEM2) and ∼8 GB (hash table) respectively, with little spatial or temporal locality. In `kmer-cnt`, there is low spatial locality because a 1-2 byte counter is updated for every 64 bytes

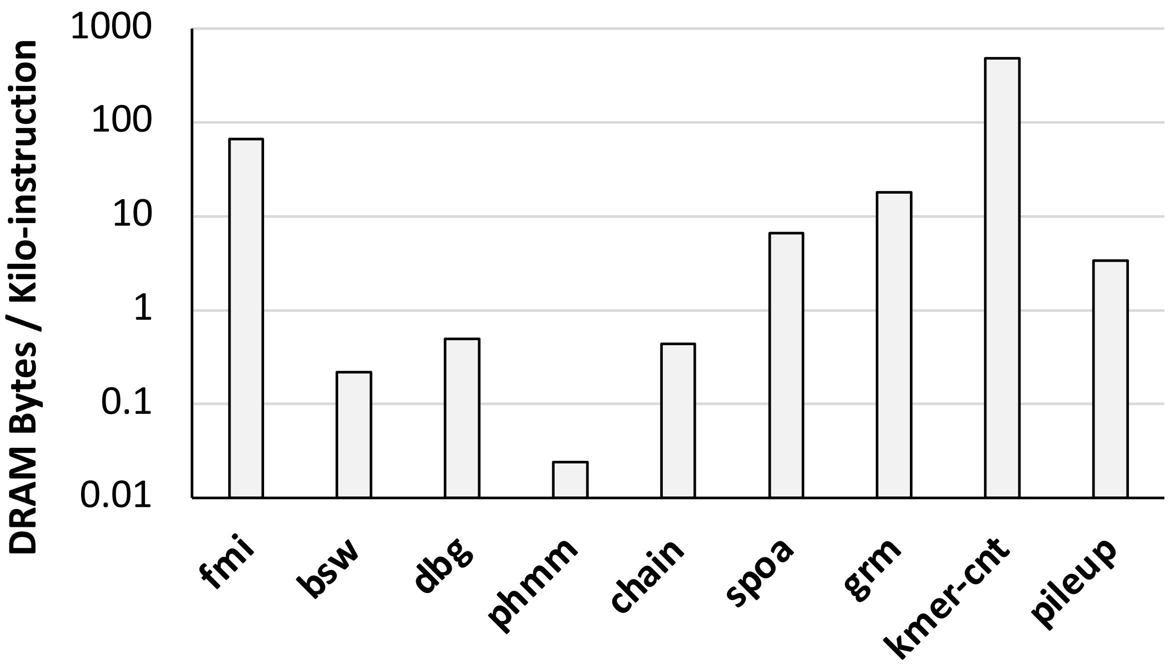


Figure 6.6: Off-chip data requirements for different benchmarks.

(cache block) read from memory. Potential approaches to improve `kmer-cnt` performance include implementing cache-friendly hashing techniques like robin hood hashing [132], and improving temporal locality since the k-mers to be inserted into the hash table are known a priori. In contrast, other benchmarks like `spoa` have modest off-chip data requirements (6.62 BPKI), while compute-intensive benchmarks like `phmm` have much lower data movement (0.02 BPKI) from off-chip memory.

Cache Miss Rates: Figure 6.7 shows the L1 and L2 cache miss rates and percentage of CPU cycles spent stalling for data. Notably in `fmi` and `kmer-cnt`, 41.5% and 69.2% of CPU cycles are spent waiting for data. While `fmi` uses all the bytes in a cache block when performing OCC table lookups, `kmer-cnt` only updates a 1-2 byte counter per LLC miss and has poor spatial locality. Apart from these two, other benchmarks spend <20% of CPU cycles waiting for data.

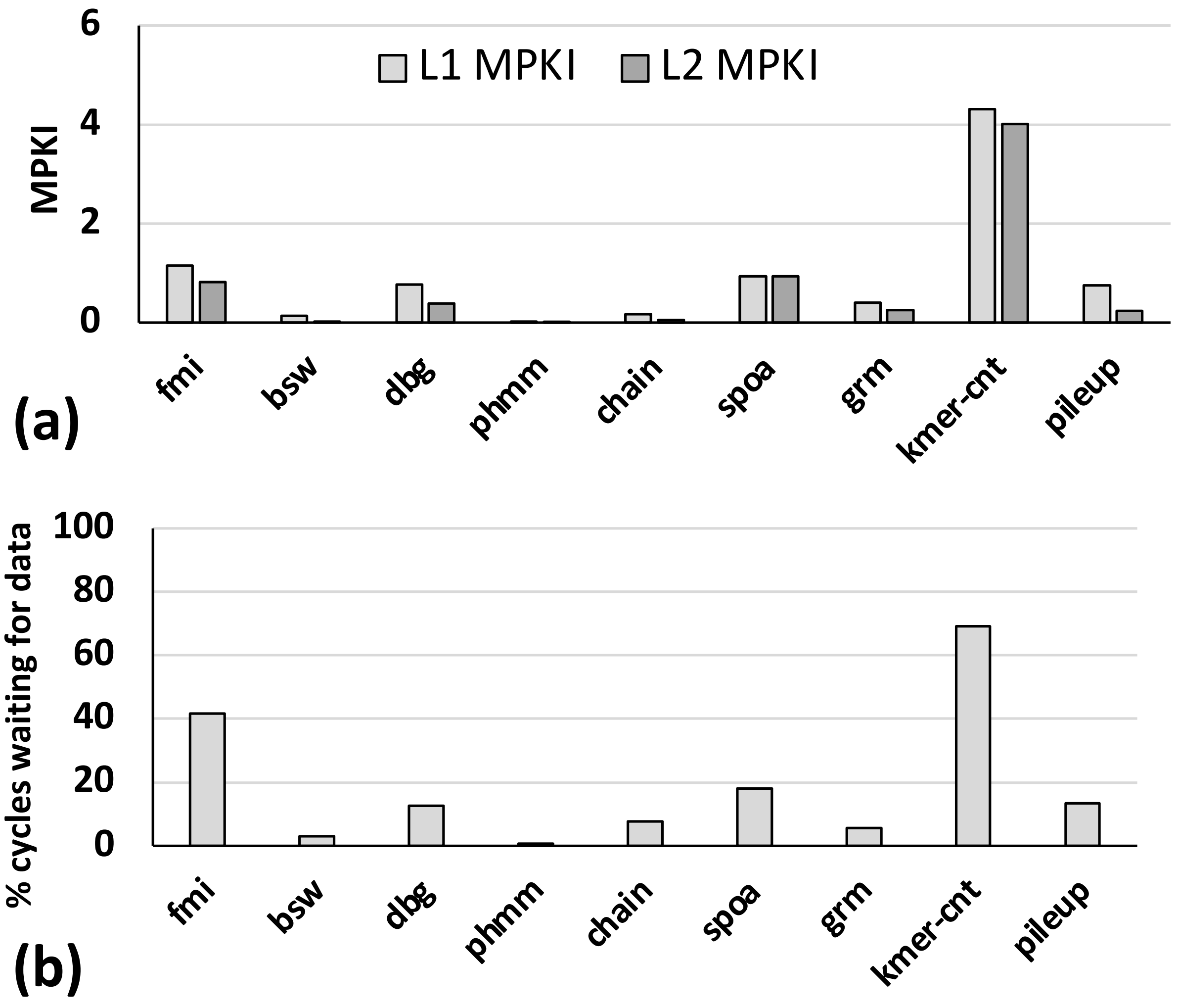


Figure 6.7: Cache misses and stalls for different benchmarks.
(a) L1 and L2 misses per kilo-instruction (MPKI) (b) Percentage of CPU cycles spent for waiting for data.

6.3.4.B GPU benchmarks

When accessing global memory, `abea` and `nn-base` kernels were unable to achieve peak memory bandwidth due to strided or irregular data accesses. This is shown in Table 6.5.

The extent of irregularity of memory accesses in both GPU kernels is a direct artifact of the type of computation performed. For `nn-base`, neural network model weights and inputs can be loaded in several large accesses to memory at the start of computation. Since Bonito's convolutional neural network is comprised of many layers of separable convolutions, these matrix vector multiplications are not too large and can be performed in shared memory. At the end, results are

	abea	nn-base
Global Load Efficiency	25.5 %	70.3 %
Global Store Efficiency	68.5 %	100 %

Table 6.5: Proportion of GPU global memory bandwidth used.

written to global memory in contiguous transactions. For the `abea` kernel, however, there are dependencies between consecutive diagonal bands of the dynamic programming matrix which are computed. In order to calculate the matrix efficiently, the previous three rows (which the following band computation is dependent on) are stored in shared memory. This leaves no room to cache the reference's k-mer current model and other frequently accessed data in shared memory. The resulting accesses to global memory are performed with sub-optimal efficiency due to the decreased spatial locality of data accesses.

6.3.5 Thread Scaling

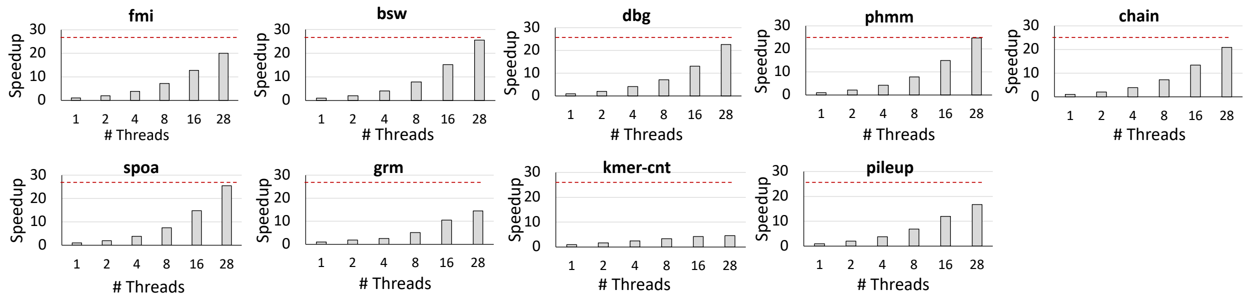


Figure 6.8: Thread scaling for different kernels in GenomicsBench.
Thread scaling for different kernels in GenomicsBench. Dotted red line shows the maximum speedup achievable on the experimental system with 28 cores. Experiments were performed on a dual socket (14-core per socket Haswell machine (Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz, AVX2) with 35 MB LLC).

Figure 6.8 shows the thread scaling behavior of the multi-threaded versions of the irregular CPU benchmarks. All inputs to these benchmarks are grouped into independent tasks with each task dynamically scheduled on a CPU thread using OpenMP. Almost all GenomicsBench benchmarks benefit from coarse-grained task-level parallelism. It can be seen that most of the benchmarks achieve perfect scaling (`bsw`, `dbg`, `phmm` and `spoa`), while `fmi` and `chain` achieve near-perfect scaling. `kmer-cnt` uses close to the peak random access memory-bandwidth on our system and does not scale well with increasing number of threads, whereas `pileup` suffers from random memory accesses.

6.3.6 Microarchitectural Bottleneck Analysis

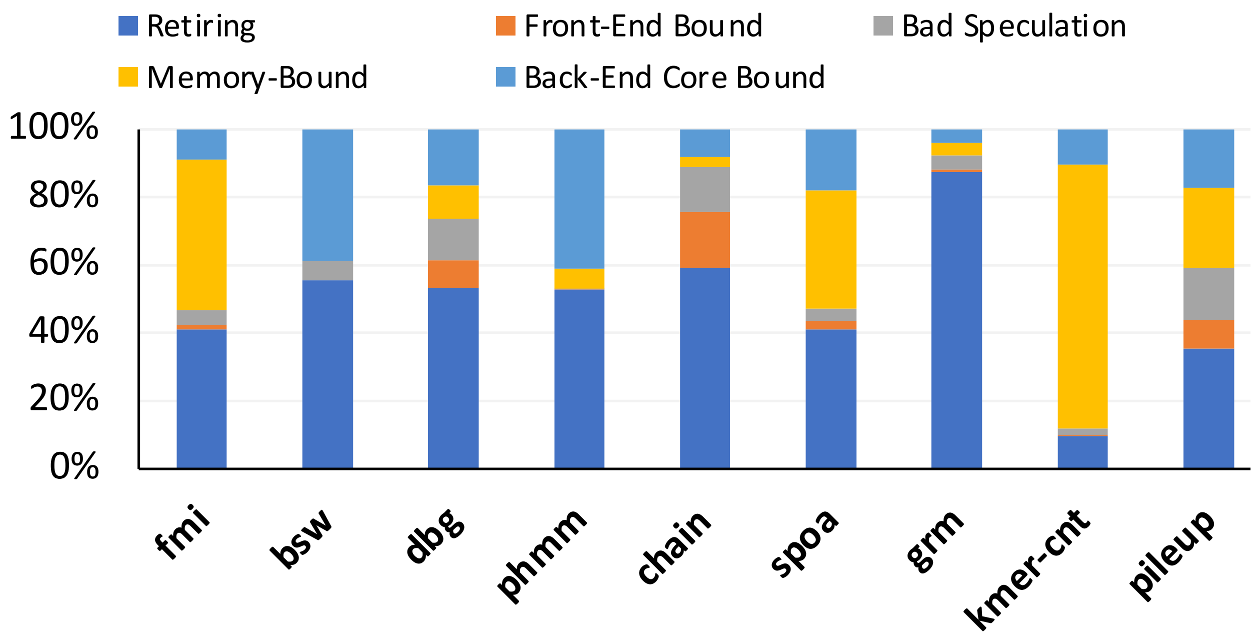


Figure 6.9: Microarchitectural bottleneck analysis.
Top-down microarchitectural bottleneck analysis of kernels (single thread).

Figure 6.9 shows the results of top-down analysis [179]. It can be seen that memory-bound benchmarks like `fmi` and `kmer-cnt` spend 44.4% and 86.6% of their pipeline slots waiting for data. For `fmi`, >80% of OCC table accesses lead to opening of a new DRAM page making the accesses highly irregular. There is also little spatial or temporal locality in k-mer counting. Each update to the k-mer count table results in a last-level cache miss leading to significant memory latency-related stalls. Some of these stalls could potentially be mitigated by implementing software prefetching [132], since the k-mers to be looked up are known in advance. Compute-intensive benchmarks like `bsw`, `chain` and `phmm` spend >50% of their pipeline slots retiring instructions. They are bottlenecked by backend core resources because of limited number of available ports for scheduling vector and floating point instructions. `grm` performs CPU-friendly dense matrix multiplication and makes best use of available CPU pipeline slots (87.70% retiring). The memory-related stalls in `spoa` and `pileup` result from cache misses during incremental update of the partial-order alignment graph and random accesses to the read alignment records respectively.

6.4 Summary

In this chapter, we discuss the GenomicsBench benchmark suite, containing 12 computationally intensive genomics kernels drawn from popular bioinformatics software tools. We perform detailed instruction level and microarchitectural analyses on these kernels to expose their performance bottlenecks. We also observe that the irregular data-parallelism in these benchmarks cannot be easily exploited by commodity hardware. GenomicsBench is open sourced to the broader research community and is available at https://github.com/arun-sub/genomicsbench.

CHAPTER 7

Conclusion

We are dealing with and producing unstructured data at an unprecedented scale today in forms such as social media posts, system logs, network packets, genome sequence data, emails and news articles. There is also increased demand for real-time analyses and prediction from unstructured data (e.g., deep packet inspection at 10-100 Gbps line rates). All these necessitate fast and efficient pattern matching approaches that can scan incoming data for interesting patterns. Conventional CPUs and GPUs, however perform poorly on pattern matching tasks that are typically characterized by irregular memory accesses and input-dependent control flow. In this dissertation, we presented several software techniques and hardware acceleration approaches to overcome the limitations of general-purpose processing for pattern matching workloads.

First, we leverage the massive bit-level parallellism of in-memory hardware accelerators to demonstrate enumerative parallelization of finite state automata (FSA) on Micron's Automata Processor (AP). Finite state automata are the widely used computational model for pattern matching. Next, building on the success of in-memory hardware accelerators for automata processing, we demonstrate that last-level SRAM-based caches of general purpose processors can be repurposed for efficient automata processing and propose the Cache Automaton architecture. We then describe two pattern matching applications in genomics, a large source of unstructured data. We present two hardware-software co-design efforts to accelerate exact string matching and approximate string matching in the computationally intensive read alignment step. Having identified that memory bandwidth is the bottleneck for the exact string matching, we design the Enumerated Radix Trees (ERT) data structure that trades off memory space for memory bandwidth. We also design custom hardware to accelerate ERT traversal. The proposed accelerator leverages fine-grained context switching to saturate memory bandwidth. For approximate string matching, we design a string-independent hardware-friendly automata, Silla and the hardware accelerator SillaX that is customized for matching genomics data by supporting features such as affine gap scoring and traceback. Together, these two accelerators can either be deployed remotely (e.g., Amazon F1 cloud) enabling wider adoption, or be directly integrated into sequencing machines to enable real-time whole genome sequencing analysis.

Finally, we present the GenomicsBench benchmark suite to characterize the computational requirements of common pattern matching tasks in different genomics pipelines. GenomicsBench contains 12 computationally intensive genomics kernels drawn from popular bioinformatics software tools. We perform detailed instruction level and microarchitectural analyses on these kernels to expose their performance bottlenecks. We observe that the irregular data-parallelism in these benchmarks cannot be easily exploited by commodity hardware, motivating the need for newer architectures to exploit irregular data-parallelism, or newer vectorization friendly algorithms for these computational tasks. GenomicsBench is under active development and is open sourced to the broader research community.

BIBLIOGRAPHY

[1] Accelerate precision medicine with microsoft genomics. https://www.azure.microsoft.com/en-us/resources/accelerate-precision-medicine-with-microsoft-genomics/.

[2] The all of us research program. https://allofus.nih.gov/.

[3] Artic network. real-time molecular epidemiology for outbreak response. https://artic.network/.

[4] Bonito. https://github.com/nanoporetech/bonito.

[5] Cache Allocation Technology. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[6] Caenorhabditis elegans 40x coverage dataset, pacific biosciences. http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans/list.html.

[7] E.coli ont data. loman lab. https://zenodo.org/record/1172816/files/Loman_E.coli_MAP006-1_2D_50x.fasta.

[8] Genome analysis toolkit: Variant discovery in high-throughput sequencing data. https://software.broadinstitute.org/gatk/.

[9] Germline short variant discovery (snps + indels). best practices workflow. https://software.broadinstitute.org/gatk/best-practices/workflow?id=11145.

[10] Huge Pages. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[11] Idc: Expect 175 zettabytes of data worldwide by 2025. https://www.networkworld.com/article/3325397/idc-expect-175-zettabytes-of-data-worldwide-by-2025.html.

[12] India to launch its 1st human genome cataloguing project. https://economictimes.indiatimes.com/news/science/india-to-launch-its-1st-human-genome-cataloguing-project/articleshow/70323116.cms?from=mdr.

[13] Intel vtune profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[14] Medaka. https://github.com/nanoporetech/medaka.

[15] Method and system to dynamically power-down a block of a pattern-recognition processor. Patent US 9389833 B2.

[16] Mgi delivers the $100 genome at agbt conference. https://www.genengnews.com/news/mgi-delivers-the-100-genome-at-agbt-conference/.

[17] Mgi unveils extreme throughput sequencing platform at agbt to enable $100 human genome. https://www.genomeweb.com/sequencing/mgi-unveils-extreme-throughput-sequencing-platform-agbt-enable-100-huma

[18] Micron Automata Processing. http://www.micronautomata.com/.

[19] Micron Automata Processing D480 Documentation Design Notes. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.

[20] Micron Automata Processing D480 Software Development Kit. AP Flow Concepts. http://micronautomata.com/apsdk_documentation/latest/h1_ap.html.

[21] Microsoft unveils genomics innovation and new partners at ashg 2018. https://cloudblogs.microsoft.com/industry-blog/health/2018/10/15/microsoft-unveils-genomics-innovation-and-new-partners-at-ashg-2018/.

[22] Nanopore sequencing accuracy. https://nanoporetech.com/accuracy.

[23] Nanopore wgs consortium data. https://github.com/nanopore-wgs-consortium/NA12878/blob/master/Genome.md.

[24] Next-generation-sequencing.v1.5.7. https://twitter.com/AlbertVilella/status/1291669705799983106.

[25] Novaseq 6000. https://sapac.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/novaseq-6000-system-specification-sheet-770-2016-025.pdf.

[26] Nvidia visual profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[27] Oxford nanopore minion. https://nanoporetech.com/products/minion.

[28] Oxford nanopore variant calling workflow (limited support release). https://github.com/kishwarshafin/pepper/blob/r0.1/docs/PEPPER_variant_calling.md.

[29] Personalised medicine at fda. a progress and outlook report. http://www.personalizedmedicinecoalition.org/Userfiles/PMC-Corporate/file/PM_at_FDA_A_Progress_and_Outlook_Report.pdf.

[30] Run the germline gatk best practices pipeline for $5 per genome. https://software.broadinstitute.org/gatk/blog?id=11415.

[31] runsa/runss custom command line analysis. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/running-runsa-runss-custom-analysis-from-the-command-line.html.

[32] Sequence alignment map format specification. https://samtools.github.io/hts-specs/SAMv1.pdf.

[33] Uk plans to sequence 5 million genomes in 5 years. https://www.bionews.org.uk/page_138891.

[34] Uscs genome browser. https://genome.ucsc.edu/.

[35] Athena Ahmadi, Alexander Behm, Nagesh Honnalli, Chen Li, Lingjie Weng, and Xiaohui Xie. Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic acids research*, 40(6):e41–e41, 2011.

[36] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[37] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 2–9. IEEE, 2005.

[38] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 175–188. ACM, 1998.

[39] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. Aspen: A scalable in-sram architecture for pushdown automata. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 921–932. IEEE, 2018.

[40] David A Bader, Yue Li, Tao Li, and Vipin Sachdeva. Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 163–173. IEEE, 2005.

[41] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154. ACM, 2007.

[42] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 2008 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, USA, November 6-7, 2008*, pages 50–59, 2008.

[43] Michela Becchi, Mark A. Franklin, and Patrick Crowley. A workload for evaluating deep packet inspection architectures. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pages 79–89, 2008.

[44] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2009.

[45] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. Entity resolution acceleration using micron's automata processor. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[46] William J. Bowhill, Blaine A. Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Oscar Mendoza, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, Jayen Desai, Jason Crop, Brian Brock, Dave Bradley, Chris Bostak, Sal Bhimji, and Matt Becker. The xeon® processor E5-2600 v3: a 22 nm 18-core product family. *J. Solid-State Circuits*, 51(1):92–104, 2016.

[47] Ashley Cacho, Ekaterina Smirnova, Snehalata Huzurbazar, and Xinping Cui. A comparison of base-calling algorithms for illumina sequencing technology. *Briefings in bioinformatics*, 17(5):786–795, 2016.

[48] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Boosting the fm-index on the gpu: effective techniques to mitigate random memory access. *IEEE/ACM transactions on computational biology and bioinformatics*, 12(5):1048–1059, 2015.

[49] Alejandro Chacón, Juan Carlos Moure, Antonio Espinosa, and Porfidio Hernández. n-step fm-index for faster pattern matching. *Procedia Computer Science*, 18:70–79, 2013.

[50] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool.

[51] Christopher C Chang, Carson C Chow, Laurent CAM Tellier, Shashaank Vattikuti, Shaun M Purcell, and James J Lee. Second-generation plink: rising to the challenge of larger and richer datasets. *Gigascience*, 4(1):s13742–015, 2015.

[52] Mau-Chung Frank Chang, Yu-Ting Chen, Jason Cong, Po-Tsang Huang, Chun-Liang Kuo, and Cody Hao Yu. The smem seeding acceleration for dna sequence alignment. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 32–39. IEEE, 2016.

[53] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. Architecting an energy-efficient dram system for gpus. In *High Performance Computer Architecture (HPCA)*, 2017.

[54] Peng Chen, Chao Wang, Xi Li, and Xuehai Zhou. Accelerating the next generation long read mapping with the fpga-based system. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(5):840–852, 2014.

[55] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. A 22nm 2.5 mb slice on-die l3 cache for the next generation xeon® processor. In *VLSI Technology (VLSIT), 2013 Symposium on*, pages C132–C133. IEEE, 2013.

[56] Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. A novel high-throughput acceleration engine for read alignment. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 199–202. IEEE, 2015.

[57] Trishul M Chilimbi, Mark D Hill, and James R Larus. Cache-conscious structure layout. In *ACM SIGPLAN Notices*, volume 34, pages 1–12. ACM, 1999.

[58] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[59] James Clark. The Expat XML parser. http://expat.sourceforge.net.

[60] J. Cong, L. Guo, P. Huang, P. Wei, and T. Yu. Smem++: A pipelined and time-multiplexed smem seeding accelerator for dna sequencing. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 206. IEEE, 2018.

[61] Subhasis Das, Tor M. Aamodt, and William J. Dally. SLIP: reducing wire energy in the memory hierarchy. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 349–361, 2015.

[62] Sutapa Datta and Subhasis Mukhopadhyay. A grammar inference approach for predicting kinase specific phosphorylation sites. *PloS one*, 10(4):e0122294, 2015.

[63] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.

[64] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[65] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[66] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC bioinformatics*, 9(1):11, 2008.

[67] Michael A Eberle, Epameinondas Fritzilas, Peter Krusche, Morten Källberg, Benjamin L Moore, Mitchell A Bekritsky, Zamin Iqbal, Han-Yu Chuang, Sean J Humphray, Aaron L Halpern, et al. A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. *Genome research*, 27(1):157–164, 2017.

[68] Yuanwei Fang, Tung Thanh Hoang, Michela Becchi, and Andrew A. Chien. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 533–545, 2015.

[69] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. Udp: A programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 55–68, New York, NY, USA, 2017. ACM.

[70] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2006.

[71] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[72] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved dfa for fast regular expression matching. *ACM SIGCOMM Computer Communication Review*, 38(5):29–40, 2008.

[73] Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: a genome sequencing accelerator. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 69–82. IEEE Press, 2018.

[74] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Seedex: A genome sequencing accelerator for optimal alignments in subminimal space. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 937–950. IEEE, 2020.

[75] Hasindu Gamaarachchi, Chun Wai Lam, Gihan Jayatilaka, Hiruna Samarakoon, Jared T Simpson, Martin A Smith, and Sri Parameswaran. Gpu accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis. *BMC bioinformatics*, 21(1):1–13, 2020.

[76] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. Hare: Hardware accelerator for regular expressions. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[77] Osamu Gotoh. Optimal sequence alignment allowing for long gaps. *Bulletin of mathematical biology*, 52(3):359–373, 1990.

[78] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[79] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–135. IEEE, 2019.

[80] Linley Gwennap. Micron accelerates automata:new chip speeds nfa processing using dram architectures. In *Microprocessor Report*, 2014.

[81] Faraz Hach, Fereydoun Hormozdiari, Can Alkan, Farhad Hormozdiari, Inanc Birol, Evan E Eichler, and S Cenk Sahinalp. mrsfast: a cache-oblivious algorithm for short-read mapping. *Nature methods*, 7(8):576, 2010.

[82] Faraz Hach, Iman Sarrafi, Farhad Hormozdiari, Can Alkan, Evan E Eichler, and S Cenk Sahinalp. mrsfast-ultra: a compact, snp-aware mapper for high performance sequencing applications. *Nucleic acids research*, 42(W1):W494–W500, 2014.

[83] Margaret A Hamburg and Francis S Collins. The path to personalized medicine. *New England Journal of Medicine*, 363(4):301–304, 2010.

[84] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain. A banded smith-waterman fpga accelerator for mercury blastp. In *2007 International Conference on Field Programmable Logic and Applications*, pages 765–769, Aug 2007.

[85] Robert S Harris. *Improved pairwise alignment of genomic DNA*. The Pennsylvania State University, 2007.

[86] Bingsheng He, Qiong Luo, and Byron Choi. Cache-conscious automata for XML filtering. *IEEE Trans. Knowl. Data Eng.*, 18(12):1629–1644, 2006.

[87] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[88] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[89] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE micro*, 27(3):63–72, 2007.

[90] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family. *J. Solid-State Circuits*, 48(8):1954–1962, 2013.

[91] Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature biotechnology*, 36(4):338–345, 2018.

[92] Peng Jiang and Gagan Agrawal. Combining SIMD and many/multi-core parallelism for finite state machines with enumerative speculation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 179–191, 2017.

[93] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[94] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović, and Rastislav Bodík. Parallelizing the web browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 7–7, Berkeley, CA, USA, 2009. USENIX Association.

[95] Carl H June, Roddy S O'Connor, Omkar U Kawalekar, Saba Ghassemi, and Michael C Milone. Car t cell immunotherapy for human cancer. *Science*, 359(6382):1361–1365, 2018.

[96] Blake Kaplan. Speculative parsing path. Bug 527623.

[97] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.

[98] Szymon M Kiełbasa, Raymond Wan, Kengo Sato, Paul Horton, and Martin C Frith. Adaptive seeds tame genomic sequence comparison. *Genome research*, 21(3):487–493, 2011.

[99] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[100] Daehwan Kim, Li Song, Florian P Breitwieser, and Steven L Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome research*, 26(12):1721–1729, 2016.

[101] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies. *BMC genomics*, 19(2):89, 2018.

[102] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[103] George Anton Kiraz. Compressed storage of sparse finite-state transducers. In *International Workshop on Implementing Automata*, pages 109–121. Springer, 1999.

[104] Shmuel Tomi Klein and Yair Wiseman. Parallel huffman decoding with applications to jpeg files. *The Computer Journal*, 46(5):487–497, 2003.

[105] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology*, 37(5):540–546, 2019.

[106] Evguenia Kopylova, Laurent Noé, and Héléne Touzet. Sortmerna: fast and accurate filtering of ribosomal rnas in metatranscriptomic data. *Bioinformatics*, 28(24):3211–3217, 2012.

[107] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome research*, 27(5):722–736, 2017.

[108] Matija Korpar and Mile Šikić. Sw#–gpu-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, 2013.

[109] Sam Kovaka, Yunfan Fan, Bohan Ni, Winston Timp, and Michael C Schatz. Targeted nanopore sequencing by real-time mapping of raw electrical signal with uncalled. *BioRxiv*, 2020.

[110] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 339–350. ACM, 2006.

[111] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[112] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357, 2012.

[113] Christopher Lee. Generating consensus sequences from partial order multiple sequence alignment graphs. *Bioinformatics*, 19(8):999–1008, 2003.

[114] Christopher Lee, Catherine Grasso, and Mark F Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[115] Chenhao Li, Kern Rei Chng, Esther Jia Hui Boey, Amanda Hui Qi Ng, Andreas Wilm, and Niranjan Nagarajan. Inc-seq: accurate single molecule reads using nanopore sequencing. *GigaScience*, 5(1):s13742–016, 2016.

[116] Heng Li. Exploring single-sample snp and indel calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012.

[117] Heng Li. Exploring single-sample snp and indel calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012.

[118] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[119] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[120] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *bioinformatics*, 25(14):1754–1760, 2009.

[121] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 373–384, 2012.

[122] Nicholas J Loman, Joshua Quick, and Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature methods*, 12(8):733–735, 2015.

[123] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 284–303. Springer, 2009.

[124] Ruibang Luo, Chak-Lim Wong, Yat-Sing Wong, Chi-Ian Tang, Chi-Man Liu, Chi-Ming Leung, and Tak-Wah Lam. Clair: exploring the limit of using a deep neural network on pileup data for germline variant calling. *Nature Machine Intelligence*, 2(4):220–227, 2020.

[125] Ruibang Luo, Thomas Wong, Jianqiao Zhu, Chi-Man Liu, Xiaoqian Zhu, Edward Wu, Lap-Kei Lee, Haoxiang Lin, Wenjuan Zhu, David W Cheung, et al. Soap3-dp: fast, accurate and sensitive gpu-based short read aligner. *PloS one*, 8(5):e65632, 2013.

[126] Medhat Mahmoud, Nastassia Gobet, Diana Ivette Cruz-Dávalos, Ninon Mounier, Christophe Dessimoz, and Fritz J Sedlazeck. Structural variant calling: the long and the short of it. *Genome biology*, 20(1):246, 2019.

[127] R. McMillen and M. Ruehle. Bioinformatics systems, apparatuses, and methods executed on an integrated circuit processing platform. `https://www.google.com/patents/US9014989`, April 21 2015. US Patent 9,014,989.

[128] Vasimuddin Md, Sanchit Misra, Heng Li, and Srinivas Aluru. Efficient architecture-aware acceleration of bwa-mem for multicore systems. In *Proceedings of the Thirty-Third IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2019.

[129] Paul Medvedev, Eric Scott, Boyko Kakaradov, and Pavel Pevzner. Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*, 27(13):i137–i141, 2011.

[130] Neil A Miller, Emily G Farrow, Margaret Gibson, Laurel K Willig, Greyson Twist, Byunggil Yoo, Tyler Marrs, Shane Corder, Lisa Krivohlavek, Adam Walter, et al. A 26-hour system of highly sensitive whole genome sequencing for emergency management of genetic diseases. *Genome medicine*, 7(1):100, 2015.

[131] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, volume 49, pages 309–328. ACM, 2014.

[132] Sanchit Misra, Tony C Pan, Kanak Mahadik, George Powley, Priya N Vaidya, Md Vasimuddin, and Srinivas Aluru. Performance extraction and suitability analysis of multi-and many-core architectures for next generation sequencing secondary analysis. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[133] P Mitankin. Universal levenshtein automata. building and properties. *Sofia University St. Kliment Ohridski*, 2005.

[134] Eugene W Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 1988.

[135] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

[136] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 529–542, 2014.

[137] Omar Naji, Christian Weis, Matthias Jung, Norbert Wehn, and Andreas Hansson. A high-level dram timing, power and area exploration tool. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 149–156. IEEE, 2015.

[138] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*, pages 182–188. IEEE, 2006.

[139] Zubair Nawaz, Muhammad Nadeem, Hans van Someren, and Koen Bertels. A parallel fpga design of the smith-waterman traceback. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 454–459. IEEE, 2010.

[140] Corey B Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L Ruzzo. Hardware acceleration of short read mapping. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 161–168. IEEE, 2012.

[141] Rachid Ounit, Steve Wanamaker, Timothy J Close, and Stefano Lonardi. Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC genomics*, 16(1):236, 2015.

[142] Alexandre Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and verification of parallel processes*, pages 196–205. Springer, 2001.

[143] Ryan Poplin, Valentin Ruano-Rubio, Mark A DePristo, Tim J Fennell, Mauricio O Carneiro, Geraldine A Van der Auwera, David E Kling, Laura D Gauthier, Ami Levy-Moonshine, David Roazen, et al. Scaling accurate genetic variant discovery to tens of thousands of samples. *BioRxiv*, page 201178, 2017.

[144] Seth H Pugsley, Arjun Deb, Rajeev Balasubramonian, and Feifei Li. Fixed-function hardware sorting accelerators for near data mapreduce execution. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 439–442. IEEE, 2015.

[145] Junqiao Qiu, Zhijia Zhao, and Bin Ren. Microspec: Speculation-centric fine-grained parallelization for FSM computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, pages 221–233, 2016.

[146] Joshua Quick, Nathan D Grubaugh, Steven T Pullan, Ingra M Claro, Andrew D Smith, Karthik Gangavarapu, Glenn Oliveira, Refugio Robles-Sikisaka, Thomas F Rogers, Nathan A Beutler, et al. Multiplex pcr method for minion and illumina sequencing of zika and other virus genomes directly from clinical samples. *Nature protocols*, 12(6):1261, 2017.

[147] Andy Rimmer, Hang Phan, Iain Mathieson, Zamin Iqbal, Stephen RF Twigg, Andrew OM Wilkie, Gil McVean, and Gerton Lunter. Integrating mapping-, assembly-and haplotype-based approaches for calling variants in clinical sequencing applications. *Nature genetics*, 46(8):912–918, 2014.

[148] Indranil Roy and Srinivas Aluru. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 13(1):99–111, 2016.

[149] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. High performance pattern matching using the automata processor. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1123–1132, 2016.

[150] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing*, page 4, 2017.

[151] Eric E Schadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human molecular genetics*, 19(R2):R227–R240, 2010.

[152] Eric E Schadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human molecular genetics*, 19(R2):R227–R240, 2010.

[153] Kishwar Shafin, Trevor Pesout, Ryan Lorig-Roach, Marina Haukness, Hugh E Olsen, Colleen Bosworth, Joel Armstrong, Kristof Tigyi, Nicholas Maurer, Sergey Koren, et al. Nanopore sequencing and the shasta toolkit enable efficient de novo assembly of eleven human genomes. *Nature Biotechnology*, pages 1–10, 2020.

[154] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[155] Martin Šošić and Mile Šikić. Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[156] Olga Spichenok, Zoran M Budimlija, Adele A Mitchell, Andreas Jenny, Lejla Kovacevic, Damir Marjanovic, Theresa Caragine, Mechthild Prinz, and Elisa Wurmbach. Prediction of eye and skin color in diverse populations using seven snps. *Forensic Science International: Genetics*, 5(5):472–478, 2011.

[157] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 259–272, New York, NY, USA, 2017. ACM.

[158] Hajime Suzuki and Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics*, 19(1):33–47, 2018.

[159] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. HAWK: hardware support for unstructured log processing. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 469–480, 2016.

[160] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.

[161] MCJ van Lanschot, LJW Bosch, M de Wit, B Carvalho, and GA Meijer. Early detection: The impact of genomics. *Virchows Archiv*, 471(2):165–173, 2017.

[162] Jan van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pages 461–472, 2012.

[163] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome research*, 27(5):737–746, 2017.

[164] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome research*, 27(5):737–746, 2017.

[165] Md Vasimuddin, Sanchit Misra, and Srinivas Aluru. Identification of significant computational building blocks through comprehensive investigation of NGS secondary analysis methods. *[Preprint] bioRXiv*, April 2018.

[166] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*, pages 105–166, 2016.

[167] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea R. Stan, and Kevin Skadron. An overview of micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*, pages 14:1–14:3, 2016.

[168] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea R Stan, and Kevin Skadron. Association rule mining with the micron automata processor. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 689–699. IEEE, 2015.

[169] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential pattern mining with the micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 135–144. ACM, 2016.

[170] Qiong Wang, Mohamed El-Hadedy, Ke Wang, and Kevin Skadron. Accelerating weeder: A dna motif search tool using the micron automata processor. 2016.

[171] Yuanrong Wang, Xueqi Li, Dawei Zang, Guangming Tan, and Ninghui Sun. Accelerating fm-index search for genomic data processing. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 65:1–65:12, New York, NY, USA, 2018. ACM.

[172] Zhen-Gang Wang, Johann Elbaz, Françoise Remacle, RD Levine, and Itamar Willner. All-dna finite-state automata with finite memory. *Proceedings of the National Academy of Sciences*, 107(51):21996–22001, 2010.

[173] Bruce W Watson. Practical optimizations for automata. In *International Workshop on Implementing Automata*, pages 232–240. Springer, 1997.

[174] Jason L Weirather, Mariateresa de Cesare, Yunhao Wang, Paolo Piazza, Vittorio Sebastiano, Xiu-Jie Wang, David Buck, and Kin Fai Au. Comprehensive comparison of pacific biosciences and oxford nanopore technologies and their applications to transcriptome analysis. *F1000Research*, 6, 2017.

[175] Ryan R Wick, Louise M Judd, and Kathryn E Holt. Performance of neural network basecalling tools for oxford nanopore sequencing. *Genome Biology*, 20(1):129, 2019.

[176] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating read mapping with fasthash. In *BMC genomics*, volume 14, page S13. BioMed Central, 2013.

[177] Hongyi Xin, Sunny Nahar, Richard Zhu, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Optimal seed solver: optimizing seed selection in read mapping. *Bioinformatics*, 32(11):1632–1642, 2015.

[178] Yi-Hua E Yang and Viktor K Prasanna. Optimizing regular expression matching with sr-nfa on multi-core systems. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 424–433. IEEE, 2011.

[179] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[180] Katherine Yelick, Aydın Buluç, Muaaz Awan, Ariful Azad, Benjamin Brock, Rob Egan, Saliya Ekanayake, Marquita Ellis, Evangelos Georganas, Giulia Guidi, et al. The parallelism motifs of genomic data analysis. *Philosophical Transactions of the Royal Society A*, 378(2166):20190394, 2020.

[181] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102. ACM, 2006.

[182] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 93–102. IEEE, 2006.

[183] Xiaodong Yu and Michela Becchi. Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 18:1–18:10, New York, NY, USA, 2013. ACM.

[184] Xiaodong Yu, Bill Lin, and Michela Becchi. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications*, 32(10):1822–1833, 2014.

[185] Matei Zaharia, William J Bolosky, Kristal Curtis, Armando Fox, David Patterson, Scott Shenker, Ion Stoica, Richard M Karp, and Taylor Sittler. Faster and more accurate sequence alignment with snap. *arXiv preprint arXiv:1111.5572*, 2011.

[186] Eleftheria Zeggini, Anna L Gloyn, Anne C Barton, and Louise V Wain. Translational genomics and precision medicine: Moving from the lab to the clinic. *Science*, 365(6460):1409–1413, 2019.

[187] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 377–384. IEEE, 2013.

[188] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. A greedy algorithm for aligning dna sequences. *Journal of Computational biology*, 7(1-2):203–214, 2000.

[189] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 619–630, 2015.

[190] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the ”embarrassingly sequential”: parallelizing finite state machine-based computations through principled speculation. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 543–558, 2014.

[191] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. Brill tagging on the micron automata processor. In *Semantic Computing (ICSC), 2015 IEEE International Conference on*, pages 236–239. IEEE, 2015.