SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Metal with Rust: High-Performance Regex Execution on Unified Memory GPUs

Marius De Kuthy Meurers

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Metal with Rust: High-Performance Regex Execution on Unified Memory GPUs

Author:	Marius De Kuthy Meurers
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Simon Ellmann, M.Sc.
Submission Date:	March 14, 2025

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Marius DKM

Munich, March 14, 2025 Marius De Kuthy Meurers

Acknowledgments

I would like to express my gratitude to my advisor Simon Ellmann, M.Sc. and my supervisor Prof. Dr. Thomas Neumann for the opportunity to work on this interesting topic.

Furthermore, many people, especially my friends have made valuable suggestions. Without your ideas and corrections, the thesis would not be as good as it is now. I want to specifically thank my parents for their support, encouragement and proofreading.

Abstract

The explosive growth of data-intensive applications demands high-performance regular expression (regex) engines capable of processing large-scale text efficiently. While GPUs offer massive parallelism, traditional architectures suffer from CPU-GPU data transfer overhead, limiting practical performance gains. This thesis addresses these challenges through the design and implementation of a GPU-accelerated regex engine optimized for Apple's M1 unified memory architecture. By eliminating data copying between CPU and GPU through unified memory, and leveraging Rust with the Metal API, we develop a novel pipeline that compiles regex patterns into minimized deterministic finite automata (DFA) for execution via a bit-parallel Shift-And-Dist algorithm on the GPU.

Key innovations include just-in-time (JIT) shader compilation to hardcode automaton masks, reducing memory latency by 48%, and optimized thread dispatching strategies that minimize idle GPU resources. Our evaluation demonstrates significant performance improvements: the GPU implementation achieves a peak throughput of 10.87 GiB/s, outperforming a multi-threaded Rust CPU baseline by up to $4\times$ for complex patterns. Unified memory eliminates traditional data transfer bottlenecks, while JIT optimizations enable sublinear scaling for inputs below 2 GiB. Beyond this threshold, memory caching effects introduce bimodal performance distributions, revealing critical architectural trade-offs. Comparative analysis shows `dispatchThreads` reduces thread overhead by 32% compared to `dispatchThreadgroups`, establishing it as the optimal dispatching method for dispatching GPU threads.

This work provides a framework for exploiting Apple Silicon's architectural innovations in computational tasks, offering insights into memory-aware GPU programming. The results highlight unified memory's potential to bridge the gap between versatility and performance in modern regex processing, with implications for databases, security systems, and bioinformatics pipelines. Our implementation opens new avenues for integrating GPU acceleration into high throughput text processing workflows.

Contents

1. Introduction

Regular expressions (regex) have become a cornerstone of modern computing, enabling efficient text processing in applications ranging from data validation to network intrusion detection. As data volumes grow exponentially, the demand for high-performance regex engines has intensified. Traditional CPU-based implementations, while versatile, often struggle to meet the throughput requirements of large-scale datasets. This challenge has spurred interest in leveraging the parallel processing capabilities of graphics processing units (GPUs), which excel at executing repetitive tasks across massive data streams. However, despite significant advancements in GPU-based regex engines, the unique architecture of Apple's M1 GPUs – particularly their unified memory model – remains underexplored, presenting both an opportunity and a gap in current research.

The motivation for this work stems from two key observations. First, while prior studies, such as Jakob et al.'s early GPU-based Snort implementation [20] and Le Glaunec et al.'s HybridSA framework [28], demonstrate the potential of GPU acceleration, they primarily target discrete GPU architectures with separate CPU-GPU memory systems. This introduces latency from data transfers, limiting real-world efficiency. Second, Apple's M1-series chips, with their unified memory architecture, eliminate this bottleneck by allowing CPUs and GPUs to access shared memory directly. Yet, no existing work comprehensively evaluates regex execution on this platform. This thesis addresses this gap by developing a regex engine optimized for the M1 GPU, combining the performance benefits of GPU parallelism with the architectural advantages of unified memory.

The primary aim of this thesis is to design, implement, and evaluate a high-performance regex engine for Apple M1 GPUs using the Metal API and Rust. The engine employs the Shift-And-Dist algorithm, an extension of the bit-parallel Shift-And method that supports variable-length transitions, enabling efficient matching of complex patterns. Key contributions include:

- A compilation pipeline converting regex patterns to minimized deterministic finite automata (DFA) and GPU-executable bitmasks.
- Just-in-time (JIT) shader optimizations to reduce memory latency.
- Evaluation of engine performance against CPU-based regex libraries.

Central to this work are several foundational concepts. The regex engine leverages regular expressions, a powerful pattern-matching syntax used in text processing. Regex patterns are compiled into deterministic finite automata (DFA) for efficient execution on the GPU. The Apple M1 GPU architecture, with its unified memory model, enables zero-copy data sharing between CPU and GPU, reducing latency and enhancing performance. The Shift-And-Dist algorithm, a bit-parallel extension of Shift-And, supports distance transitions, balancing flexibility and performance on GPU hardware.

Related work spans two decades, beginning with Jakob et al.'s fragment shader-based approach [20] and evolving to modern frameworks like iNFAnt [8] and HotStart [30]. Recent breakthroughs, such as HybridSA [28], achieve 11 times speedups over CPU implementations by dynamically partitioning workloads between CPU and GPU. However, these solutions are tailored to traditional GPUs and do not exploit unified memory. This thesis builds on these foundations while introducing optimizations specific to Apple's architecture.

The remainder of this thesis is structured as follows: Chapter 2 reviews related work in GPU-based regex processing. Chapter 3 details the Apple M1 GPU architecture, Metal API, and regex theory. Chapter 4 presents the implementation of the regex engine, including shader optimizations and memory management. Chapter 5 evaluates performance between the different execution modes and against CPU-based engines. Finally, Chapter 6 concludes with insights and future research directions.

By bridging the gap between regex theory, GPU programming, and Apple's hardware innovations, this work advances the state of the art in high-performance pattern matching, offering a blueprint for leveraging unified memory architectures in computational tasks.

2. Related Work

The concept of using coprocessors such as graphics cards for pattern matching was first proposed approximately 20 years ago by Jakob et al., who sought to overcome CPU-bound limitations in network packet inspection for intrusion detection systems (IDS) [20]. Their work was motivated by performance analysis of Snort, a widely-used open-source IDS, which revealed that string matching operations consumed approximately 60% of Snort's total runtime. During that time, frameworks for GPU programming weren't as accessible for compute loads as today. Their proof of concept repurposed fragment shaders to run a parallel Knuth-Morris-Pratt algorithm [25] for string matching. Their shaders work by sliding the text across the pattern and recording matches. If any matches occur, the fragment shader will write to the framebuffer, which is then read back by the CPU using an occlusion query to determine if a match occurred. Their results show that they can outperform conventional Snort by up to 40%, however, their implementation, *Pixelsnort*, degraded under high CPU loads, converging to the speed of Snort. Moreover, they state that their implementation is not optimal for using the full potential of the GPU.

The availability of more software development kits (SDKs) like CUDA and OpenCL and the rise of general-purpose GPU (GPGPU) programming has made it easier to offload compute tasks to the GPU. This led Vasiliadis et al. to develop *Gnort* [43]. Gnort is a high-performance network intrusion detection system (NIDS) based on Snort implementation that utilizes an NVIDIA GeForce 8 Series (G80) GPU using the CUDA framework. A SIMD-optimized version of the Aho-Corasick algorithm [2] was implemented for asynchronous pattern matching. Their evaluation demonstrated that Gnort can outperform Snort by a factor of two in real-world scenarios and have a processing throughput of 2.3 GB/s using synthetic network traces on a single G80 GPU.

In follow-up work, Vasiliadis et al. [44] developed a DFA-based regex engine by compiling patterns to non-deterministic Finite Automata (NFAs) and then to deterministic finite automata (DFAs) for later executing the state machines on the GPU. Each stream processor on the GPU is responsible for processing one network packet at a time, allowing for a 48 times speedup over traditional CPU implementations with a raw throughput of 16 Gbit/s, a 60% increase in Snort packet processing speed.

On the same hardware, Smith et al. developed a SIMD algorithm for network signature matching [40]. They implemented two different state machines: deterministic finite automata (DFAs) and extended finite automata (XFAs). XFAs are extended DFAs that include auxiliary memory for counters, enabling the XFAs to have fewer states than the equivalent DFA, reducing the memory footprint. However, XFAs introduce branching that worsens SIMD performance due to divergent execution paths. This is reflected in their experiments, which compared to the execution on an Intel Pentium 4 CPU, the XFAs only are 6.7 $\times$ faster while the DFAs are 8.6 $\times$ faster.

Using NFAs for regex matching on GPUs, Cascarano et al. in their *iNFAnt* paper [8] mitigate the state explosion problem when converting complex regex to a DFA. The design allows storing large and complex regexes in a small amount of memory. Additionally, their implementation uses N-multistriding, meaning that N characters get combined into one. This combination of characters is done on the CPU side by iterating over the input and translating all pairs into a single output symbol using a hashed lookup table. Multistriding has the overall effect that the input to the GPU is shorter, requiring fewer iterations of the traversal algorithm, which is the bottleneck with a large amount of instructions per iteration. Without the multistriding, they achieve similar speeds to the state-of-the-art on the CPU, but using 2-striding, they outperform by roughly double the speed.

Improving the NFA techniques Zu et al. [47] are able to achieve a speedup of $29 \sim 46$ over iNFAnt topping out at a matching speed of 10 Gbit/s compared to only 0.26 Gbit/s. Seeing that Cascarano et al. NFAs have a lot of simulations and active transitions, resulting in higher computational cost and higher thread divergence, Zu et al. optimized their NFAs to have as few states active as possible. The combination of states into groups that cannot be active at the same time reduces the thread divergence further. Their final optimization is the use of a packet interweaving technique that mixes packet slices into 128 B blocks to better utilize GPU's memory and cache.

Besides using GPUs for matching packets in intrusion detection systems, they were used to compare biological sequences in DNA. Sandes et al. [38] used GPUs in their paper *CUDAlign* to execute the Smith-Waterman (SW) algorithm that compares two Megabase genomic sequences. Their implementation was able to run a workload in 21 hours, which took over 4 days on a 64-processor cluster.

For database applications, Sitaridi et al. [39] accelerate the SQL `LIKE` operator using GPUs. Their research found that uncoordinated memory requests impacted performance significantly, leading them to use the Knuth-Morris-Pratt (KMP) algorithm. Together with a different memory layout of the input called pivoting, meaning storing N-characters of each string next to each other instead of the whole string, they further optimize predictable memory access patterns. Their implementation resulted in a much higher throughput on the TPC-H Query 16 of 98.7 GB/s compared to the RE2 CPU implementation with only 40.8 GB/s.

Liu et al. [30] identify two critical challenges in previous NFA implementations: excessive data movement and poor compute utilization. Their solution, *HotStart*, addresses these issues through three innovations. First, they introduce a space-efficient format for storing NFA matchsets and topology information, reducing storage requirements to just 1% of the previous solutions. Matchsets specify the symbols that each state in the NFA can match. Second, they compress matchsets by converting indirect memory reads to pure range check computations using boolean flags that indicate whether a matchset is complete or complementary, replacing approximately 70% of the state matchset lookups with efficient range checks. Third, they implement a hybrid execution approach that employs a one-to-one mapping for frequently active "hot" states (identified through profiling the first 1KB of input) and a worklist-based approach for rarely active "cold" states. Their evaluation demonstrates a 20.5 times speedup over iNFAnt, establishing HotStart as a significant advancement in GPU-based regex matching.

In their subsequent research, named Asynchronous Parallel Automata Processing (*AsyncAP*), Liu et al. [29] address the critical issue of GPU resource underutilization in automata processing, mainly when parallelism is limited. AsyncAP exploits an overlooked dimension of parallelism inherent in the input symbols. By enabling GPU threads to process input streams asynchronously from different starting points, AsyncAP substantially improves throughput and scalability. This approach effectively overcomes the limitations of conventional GPU-based automata processing engines that struggle to fully utilize GPU compute resources in parallelism-constrained scenarios. The strength of AsyncAP lies in its adaptability across varying degrees of parallelism, maintaining high efficiency regardless of task scale. Evaluation results demonstrate AsyncAP's superior performance, achieving a 2.4 $\times$ speedup over GPU-NFA, the state-of-the-art in the paper. For tasks with limited parallelism, AsyncAP delivers even up to 57.9 $\times$ performance improvement by leveraging otherwise idle GPU resources.

The most recent approach for GPU regex execution, and indeed the inspiration for this thesis, is the *HybridSA* framework developed by Le Glaunec et al. [28]. As the title suggests, their research implements a hybrid approach leveraging both GPU and CPU resources for concurrent multi-pattern matching. Their method efficiently executes NFAs on the GPU by extending the bit parallelism algorithm Shift-And. The authors enhance the standard Shift-And algorithm with three significant extensions: Shift-And-Dist, Shift-And-Gap, and Shift-And-Opt. We will explore the Shift-And-Dist algorithm in detail in Section 3.4.1. The HybridSA runtime dynamically selects the optimal algorithm based on the regex pattern complexity, deploying complex patterns to the CPU while executing simpler patterns on the GPU. Their comprehensive evaluation demonstrates impressive performance advantages, outperforming the previously mentioned HotStart implementation by Liu et al. by a factor of at least 11. HybridSA outperforms Hyperscan, the fastest SIMD-CPU implementation, by at least 4 $\times$, setting a new high-performance regex matching benchmark.

3. Background

This chapter will give an overview of the unified memory architecture of the Apple M1 GPU Chip, the Metal API, how to interface it using Rust, and what a Regex is. We will focus only on the Apple G13 GPU architecture, which has the Apple 7 GPU feature set and is used on the Apple A14 and M1, including M1 Max and Ultra Chips.

3.1. Apple G13 GPU Architecture

A graphics processing unit (GPU) is specialized hardware designed to perform parallel computations, specifically for graphics. However, over the last decade, they have become powerful co-processors that calculate specific arithmetic instructions much more efficiently in a massively parallel manner [5].

The cores of an Apple GPU at its base do not differ from the CPU cores in transistor count and I/O bus width. The distinctions in the GPU cores are the $\frac{1}{3}$ lower clock speed but 8 times higher parallelism, resulting in an about 2.67 times speedup in performance per core on 100% utilization. The speedup is because they have a much larger number of arithmetic logic units (ALUs), 128 on the GPU vs. about 14 on the performance cores on the CPU [23, 42, 11]. But to utilize this amount of parallelism, the GPU needs to achieve a much higher arithmetic intensity, meaning a lot more calculations per byte, perfect for its original task of rendering graphics but also for some computational tasks like machine learning, image processing, and other tasks that require a lot of similar simple calculations.

The Apple G13 GPU is a tile-based deferred rendering (TBDR) GPU. TBDR is a rendering technique that optimizes performance and power efficiency. The rendering is done by dividing the render destination into a grid of smaller regions called tiles. The GPU processes each tile with one of its GPU cores, often running many at the same time. First, all the underlying geometry is processed, and then, in the second step, the tiles are rendered. The two-step process allows the GPU to discard any occluded primitives and only render the visible ones. A big advantage over immediate-mode (IM) GPUs, which fully process primitives, such as lines and triangles, regardless of whether they are visible in the rendering [41, 18].

A significant feature of the Apple GPUs is that they have a **unified graphics and compute architecture**, which means that the same GPU cores can be used for graphics and compute tasks, allowing for pipelines that mix graphics and compute shaders. Sharing the same hardware is a key factor for why the Apple M Chips are so power efficient since there are no separate compute cores idling and eating away power. More importantly, this allows the use of the whole hardware for compute tasks [18].

Additionally, what sets the Apple GPU apart from other laptop and desktop GPUs is the **unified memory architecture** that is shared with the CPU. In this case, unified means that the GPU and CPU can access the same memory, which is a significant advantage for data sharing. Traditional GPUs have their separate memory, which requires data to be copied between the CPU and GPU memory, introducing latency, especially for tasks requiring data to be read back from the GPU. The unified memory architecture allows for zero-copy data sharing between the CPU and GPU, allowing the GPU to directly do computational workloads on data in the system DRAM, improving performance and power efficiency [3]. Combined with the fast DRAM speeds of the *LPDDR4X* memory at 4266 MHz allow for transfer speeds of up to 68.3 GB/s. All newer Apple M-Chips, including the M1 Pro, Max, and Ultra, have newer *LPDDR5X* memory at 6400 MHz allowing for transfer speeds of up to 204.8 GB/s and on the Max with an access width of 512 bit up to 409.6 GB/s respectively [10].

CPU and GPU cores have multiple levels of cache, which are small, fast memory units that store frequently accessed data even closer to the cores than the DRAM. **Cache** wise only the level 3 (L3) 8 MB system level cache is shared with the CPU and GPU; the L1 and L2 caches are separate. The L1 cache per core is just 8 kB with and with only 1 MB of L2 Cache with bandwidth of 400 GB/s. In contrast, the L2 cache of the high-performance Firestorm CPU cores is 12 MB large, which helps absorb the bulk of CPU-side memory access since CPUs typically do less memory-intensive workloads. When transferring data between the CPU and GPU, the data always goes all the way to the memory controller and then DRAM and is not cached by the system-level cache, even though it is shared. However, the fast DRAM speeds do a good job of hiding the latency. Intel, on their Alder Lake CPUs with integrated GPU, has a shared L3 cache with the CPU, allowing transfers under 2 kB to be done about 10 times faster than on the Apple M1 Chip. [27, 26]

Below the hood, the Apple G13 GPU on the M1 has seven or eight cores, depending on the model, with each core having 32 parallel execution lanes, each called a **thread** in Metal terminology, on a CPU, they would be called a *SIMD-lane*. The equivalent of a CPU *thread* on the GPU is a **SIMD-group**. SIMD-groups are then further organized into threadgroups. Thus, the Metal hierarchy is thread $\subseteq$ SIMD-group $\subseteq$ threadgroup. In practice, the developer does not define which threads are in one SIMD-group. Instead, the dimensions of one threadgroup are set so that the threads fit seamlessly into N SIMD-groups. Each Metal SIMD-group has a stack pointer, a program counter, a 32 bit execution mask, and up to 128 general-purpose registers. Each thread can access them as 32 bit registers or as 16 bit upper or lower halves. Additionally, 256 32 bit registers are shared by all threadgroups. [21] As shown in Figure 3.1, an image would be rendered by dividing the work among threadgroups, with each thread processing a distinct pixel. This setup enables efficient memory access within each threadgroup and minimizes latency during data transfers. [9]

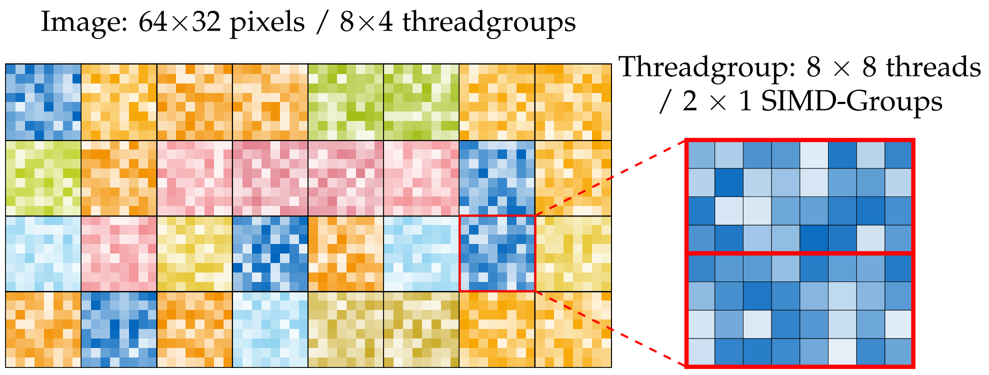


Image: 64×32 pixels / 8×4 threadgroups
Threadgroup: 8 × 8 threads / 2 × 1 SIMD-Groups

Figure 3.1.: Example of image rendered using 8×4 threadgroups, with each threadgroup consisting of two SIMD-group containing 8×4 threads.

3.1.1. Metal

To interface with the Apple GPUs in operating systems like macOS, iOS, or tvOS, Apple introduced the **Metal Framework** in 2014. Metal is a low-level, low-overhead hardware-accelerated graphics and compute API that provides near-direct access to the GPU. It is designed to provide developers with a low CPU overhead for execution workloads on the GPU. [1]

Metal provides a unified API for graphics and compute operations, allowing developers to use the same tools and techniques for both tasks. A single interface is convenient for applications that require a mix of graphics and compute workloads, such as games, simulations, and machine learning applications.

3.1.2. Metal Shading Language

The code for execution on the GPU is written in the **Metal Shading Language** (MSL), a C++-based language also compiled and optimized using clang and LLVM [32]. Shaders are small programs that run on the GPU and are used to perform calculations for rendering graphics or processing data. MSL provides a range of features for writing efficient and flexible shaders, including support for functions, control flow, and data structures. Based on C++14, it offers standard features like function overloading, templating, and preprocessing directives, making complex shader development possible Meanwhile, lambda expressions, derived classes, and exception handling are not supported. For the different types of workloads, MSL has three main function types. `Vertex` and `fragment` functions are used for different steps in the rendering pipeline. For this project, `kernel` functions are relevant. These are compute functions that can be executed over a 1-, 2-, or 3D grid using threadgroups. Besides these three basic function types, MSL supports more specialized ones like ray tracing.

Metal also includes various tools and libraries to help developers optimize their applications for performance and efficiency. Instead of the C++ standard library, it has its own `metal_stdlib`. XCode includes tools for profiling and debugging GPU workloads and libraries for common tasks such as image processing and machine learning.

Overall, Metal provides a powerful and flexible framework for leveraging the capabilities of Apple GPUs, allowing developers to create high-performance applications that take full advantage of the hardware.

3.1.3. Metal Execution Model

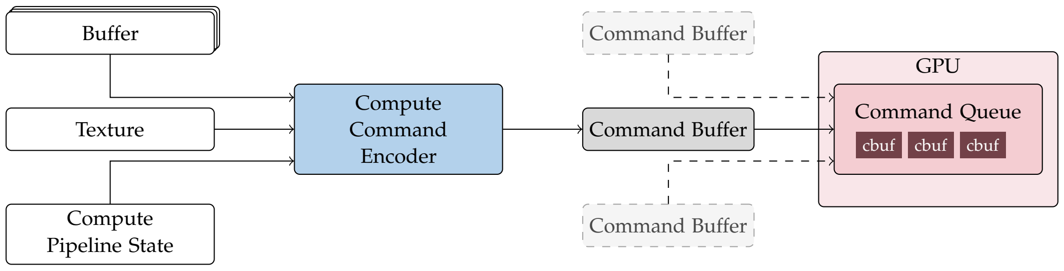


Figure 3.2.: Object relations and flow of a Metal pipeline

Figure 3.2 shows the hierarchy of executing workflows on a GPU using Metal. A `MTLDevice` represents a single GPU. It has different methods for creating objects and interacting in association with that GPU. In order to run workloads, one needs a **command queue** (`MTLCommandQueue`), which consists of one or multiple **command buffers** (`MTLCommandBuffer`), which are then eventually committed for execution on the device. A command buffer is fed by **command encoders**, which come in different kinds, for the different workloads: `MTLRenderCommandEncoder` for graphics rendering, `MTLComputeCommandEncoder` for computation, `MTLBlitCommandEncoder` for memory management and `MTLParallelRenderCommandEncoder` for multiple graphics rendering tasks encoded in parallel. In our case, only the compute command encoder is relevant since it will not be doing any rendering. [31]

For a metal shader to be encoded, it must first be compiled. Compilation can either be done during compile time or runtime. Popular machine learning libraries like TensorFlow take advantage of that to construct shaders for specific models[1]. Compiled shaders are stored in a `MTLLibrary` object, from which a `MTLFunction` can be extracted and then encoded by a command encoder.

Data has to be wrapped into `MTLBuffer`, which can then be likewise encoded into the command buffer using the command encoder. Where the memory resides can be decided using storage modes. In our case, we have unified memory between CPU and GPU, allowing us to use the `MTLStorageModeShared`.

3.2. Regular Expressions

Regular expressions, "regex" or "regexp" for short, are flexible and powerful patterns used for matching sequences of text within a larger body of text. They are based on regular languages, which are well-founded in formal language theory and have many proven relevant properties (regular languages are closed under various operations such as intersection, union, and complement). Processing regular languages also exhibit excellent runtime characteristics, with linear time complexity relative to input length, regardless of the rule set size.

Regular expressions (which, depending on implementation, may offer extensions beyond regular languages) are available in virtually all programming languages and many tools, as they serve numerous practical applications. They are commonly used to search files with tools like `grep`[14], search and replace strings in text editors, or input validation of emails or phone numbers in website registration forms.

Some commonly used regexes are:

- **Date**: `\d{2}.\d{2}.\d{4}`
- **24-Hour time**: `([01]?[0-9]|2[0-3]):[0-5][0-9]`
- **Emails**: `[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}`

Regex come in different flavors and standards, such as POSIX, PCRE (Perl Compatible Regular Expressions), ECMAScript (JavaScript), and Rust regex. As background for our investigation, we here focus on a subset of the Rust regex library[37] since it supports the most common features and comes with a parser that compiles to a simple

[1] `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/delegates/gpu/metal_delegate.mm`

High-Level Intermediate Representation (HIR) that we can use to compile our GPU matchers.

Each flavor has its own syntax and features, but the core concepts remain the same. They consist of **literals** matching their character in the text, like "a", and **metacharacters**, which have a special meaning, like "*". All the metacharacters supported by the Rust regex library are listed in the Appendix in Table A.1.

3.3. Finite Automata

Regular expressions represent a formal language that a finite automaton can recognize. **Finite automata** (FA), also known as finite state machines, are theoretical computational models used in computer science. They are abstract machines that can be in exactly one of a finite number of states at any given time, and they transition between states based on input symbols. They are used in a variety of applications, including designing lexical analyzers for compilers, verifying protocols and algorithms, and, of course, processing regular expressions.

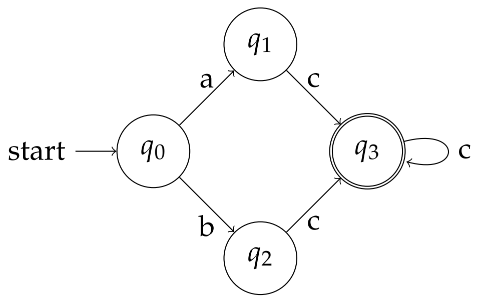
Finite automata are defined formally as a 5-tuple, (Q, Σ, T, q_0, F) consisting of:

- a finite set of states Q,
- a finite set of input symbols Σ,
- a transition function $T : Q \times \Sigma \to Q$,
- an initial state $q_0 \in Q$, and
- accept states $F \subseteq Q$.

Finite Automata can be represented as a state transition diagram. Figure 3.3 shows a state transition diagram where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b, c\}$, $T = \{(q_0, a, q_1), (q_0, b, q_2), (q_1, c, q_3), (q_2, c, q_3), (q_3, c, q_3)\}$, q_0 is the initial state and $F = \{q_3\}$.

In the state transition diagram, states are represented by circles. A finite automaton has two special states: the "start'; state q_0 and the "accept" states $F = \{q_2, q_3\}$. Start states are depicted with lone arrowheads pointing at them and accept states are drawn as a double circle. Transitions between states are represented by arrows labeled with the input symbol that triggers the transition. In the example in Figure 3.3, the automaton accepts strings that start with either "a" or "b" followed by one or more "c"s.

Such an automaton reads in a string of input symbols one at a time and transitions between states based on the input symbol. If the automaton ends in an accept state after reading the entire input string, the string is considered to match the regular expression represented by the automaton.



$$\begin{tikzcd}
 & q_1 \arrow[dr, "c"] & \\
\text{start} \arrow[r] & q_0 \arrow[u, "a"] \arrow[d, "b"'] & q_3 \arrow[loop right, "c"] \\
 & q_2 \arrow[ur, "c"'] &
\end{tikzcd}$$

Figure 3.3.: Example of a finite automaton for the regular expression $(a|b)c+$

There are two main types of finite automata: **Deterministic Finite Automata (DFA)** and **Nondeterministic Finite Automata (NFA)**. In DFAs, each state and input symbol has exactly one transition to the next state. Meanwhile, NFA can have multiple possible next or no next states for each state and input symbol. NFAs can also have ε-transitions, which allow moving to another state without consuming an input symbol. The differences have the effect that NFAs can simultaneously be active in multiple states.

In general, NFAs are more straightforward to construct from a regex but can be less efficient to execute compared to DFAs. However, both NFAs and DFAs are equivalent in terms of the languages they can recognize. Later Section 4.1, we will discuss how to convert a regex first to an NFA, then to a DFA, and then further minimize the DFA to finally compile it to a state machine that can be executed on the GPU.

3.4. Shift-And Algorithm

The **Shift-And** Algorithm was first implemented in `agrep`, a Unix tool for approximate pattern matching, by Wu and Manber in 1992 [45]. It was based on the original work of Baeza-Yates and Gonnet on the **Shift-Or** algorithm, which was published in the same journal issue in 1992 [6]. The **Shift-Or** Algorithm represents an innovative technique in string matching that leverages bit-parallelism for efficient pattern searching. What distinguishes this algorithm is its representation of search states as a bit vector of length m, with each different state represented as bits that track matching progress.

By implementing the search state as a numerical value, each search step can be executed through a minimal set of bitwise operations, shifts, and additions, eliminating the need for text buffering. This approach proves particularly effective for shorter patterns, with empirical results showing comparable performance to established algorithms like Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM). [46] The performance comes in part from the linear time complexity O(n) when the pattern length is smaller than or equal to the computer's word size. The algorithm's elegant combination of simplicity and efficiency, alongside its suitability for hardware implementation, makes it a significant contribution to the field of computer science. The Shift-And algorithm is a further development of the Shift-Or algorithm, which uses the same principles but with an inverted search state, and thus, the operations are inverted as well.

Algorithm 1 Shift-And algorithm

```
Require: $masks\_char[]$ // an array of bit vectors representing the transitions
Require: $mask\_initial$ // a bit vector representing the start states
Require: $mask\_final$ // a bit vector representing the final states
Require: $text$ // the input text
  $state \leftarrow 0$
  $n\_matches \leftarrow 0$
  for $c$ in $text$ do
      $next \leftarrow (state \ll 1)$ OR $mask\_initial$
      $state \leftarrow next$ AND $masks\_char[c]$
      if $(state$ AND $mask\_final) \neq 0$ then
          $n\_matches \leftarrow n\_matches + 1$
  return $n\_matches$
```

A Shift-And algorithm, as shown in Algorithm 1, consists of four main components: the search state *state*, the character transition masks $masks\_char[]$, the start mask *mask_initial* and the final mask $mask\_final$. It works by iterating over the input text, updating the search state based on the character read, and then checking if the search state is in a final state. Updating the search state is done by first calculating the possible transitions from the current state and then masking the possible transitions with the input character.

The search state is constructed by placing all states $q_0, q_1 \ldots q_n$ in a line, where each state is represented by a bit in a binary number. Next, each transition from q_i to q_{i+1} represents a bit shift operation. Only transitions from $(i) \rightarrow (i+1)$ are possible, which implies that no backward edges or longer jumps are supported.

Let us take the Regex r = `[ab]c|ce?` with the input `bce` as an example. The corresponding state machine is shown in Figure 3.4. Since we have two branches of the algorithm, we have two start states, q_0 and q_3. Furthermore, the final states are q_2, q_4 and q_5. To construct the search state, we do not need to save the start states in the search state since they are always active. Thus, the first states that we save in `state` are q_1 and q_4 which, when active, are represented as `0001` and `0100` respectively. When we read in the character `b`, we should only get to state q_1, therefor we save the transition from q_0 to q_1 as `0001` in `masks_char['b']`. Since we can always apply this transition,

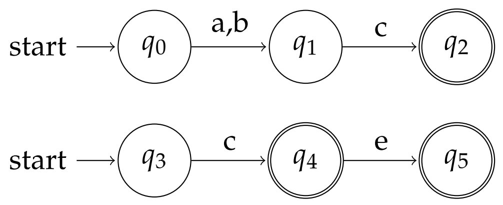


Figure 3.4.: Finite automata for the regex `[ab]c|ce?` using two start states.

we store the start transitions as `0001` and `0100` in `mask_initial` = `0101`. The final states are q_2, q_4 and q_5 which are represented as `0010`, `0100` and `1000` respectively saved in `masks_final` as `1110`.

To understand the algorithm better, let us walk through the example. We start by reading in the character `b`. *next* is calculated as:

$$next \leftarrow (state \ll 1) \text{ OR } mask\_initial$$
$$\texttt{0101} \leftarrow (\texttt{0000} \ll 1) \text{ OR } \texttt{0101}$$

The application of *mask_initial* means that independently of the input character, we could transition to the states q_1 and q_4. Then, we update the state by masking the possible transitions with the input character.

$$state \leftarrow \text{next AND } masks\_char[\text{'}b\text{'}]$$
$$\texttt{0001} \leftarrow \texttt{0101} \text{ AND } \texttt{0001}$$

Now, after the first character, we are in state q_1 as designated by the active first bit in *state*. Next, we read in `c`, calculate the next state, and update the state.

$$next \leftarrow (state \ll 1) \text{ OR } mask\_initial$$
$$\texttt{0010} \leftarrow (\texttt{0001} \ll 1) \text{ OR } \texttt{0101}$$
$$state \leftarrow \text{next AND } masks\_char[\text{'}c\text{'}]$$
$$\texttt{0010} \leftarrow \texttt{0010} \text{ AND } \texttt{0010}$$

Now we are in state q_2 and q_4. If we now compare with the final mask *states* AND `mask_final` $\neq 0$, we see that we have a match and can increment the counter.

In the next loop iteration, we read in the final character `e` and calculate the next state and update the state.

$$\texttt{1101} \leftarrow (state \ll 1) \text{ OR } mask\_initial$$
$$\texttt{1000} \leftarrow \text{next AND } masks\_char[\text{'}e\text{'}]$$

The state indicates now, that we are in state q_5. Masking, *state* with *mask_final* shows us, that we have another match and should increase the counter.

3.4.1. Shift-And-Dist

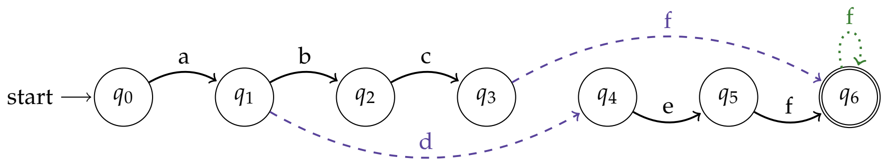


Figure 3.5.: Finite automata for the regex `a(bc|de)f*` with distance transitions.

The Shift-And-Dist algorithm is a further development of the Shift-And algorithm by Le Glaune et al. [28] to allow for longer distance transitions. By having a distance of d between the states, the algorithm allows for transitions of length $0 \leq d$. This enables new types of regexes to be matched, such as `a{0,3}b` (matching b, ab, aab and $aaab$) or `a(bc|de)f*` (matching $abcf$, $adef$, and $adefff$). Looking at the example in Figure 3.5, we can see that the distance between the state between q_1 to q_4 and q_3 to q_6 is 4, and the distance of the self-loop on q_6 is 0.

The addition to the Shift-And algorithm is done by adding a vector of bit masks for each distance d named $masks\_dist[d]$. The possible next states are then calculated by masking the current state with the corresponding distance mask to determine which states can move the distance d. Then, the possible transitions are calculated by shifting the current state by d. After computing this for all distances d, the results are then masked by the input character to determine the next state. The pseudocode for the Shift-And-Dist is shown in Algorithm 2.

Algorithm 2 Shift-And-Dist algorithm

```
Require: $masks\_char[]$ // an array of bit vectors for the transitions
Require: $mask\_initial, mask\_final$ // bit vectors for the start, final states
Require: $masks\_dist[]$ // an array of bit vectors for each distance
Require: $text$ // the input text
  $state \leftarrow 0$
  $n\_matches \leftarrow 0$
  for $c$ in $text$ do
      $next \leftarrow mask\_initial$
      for $d$ in $0 \ldots \text{len}(masks\_dist)$ do
          $next \leftarrow next$ OR $(state$ AND $masks\_dist[d]) \ll d$
      $state \leftarrow next$ AND $masks\_char[c]$
      if $(state$ AND $mask\_final) \neq 0$ then
          $n\_matches \leftarrow n\_matches + 1$
  return $n\_matches$
```

4. Implementation

We will now present the implementation of the regex engine on the Apple M1 GPU: First, the general structure of the engine, then the individual parts of the engine, and finally, how they are implemented and their optimizations.

For the implementation of the regex engine, we will use the Rust programming language. Rust is a systems programming language that is known for its performance and safety. It has a strong type system that prevents many common programming errors at compile time. Furthermore, this approach enables us to directly compare the performance characteristics of specialized hardware against the traditional CPU-based rust regex crate.

In order to interact with the Metal-based GPU through Rust, there are multiple crates. *Crates* are like self-contained libraries of Rust code, offering ready-to-use functionality that developers can import and utilize in their own programs, and are published on the Rust package registry, *crates.io*.

First **rust-gpu** [36] a project which translates Rust code to Vulkan compute shaders, which can be run using *MoltenVK*, a translation layer between Vulkan and Metal. Yet, this approach is unsuitable for our use case because of the translation overhead.

The second option is **webgpu**, "A cross-platform, safe, pure-Rust graphics API" [13], which uses the WebGPU Shading Language (WGSL) to natively run Metal shaders. After some trials with this crate, some issues were found with the GPU data buffers, which, due to compatibility with other GPU platforms, are not as fast as the native Metal buffers.

The webgpu crate currently uses the **metal-rs** [12] crate as the backend for the Metal API. Thus, this could be a third option to use the metal-rs crate directly. However, the maintainers themself recommend another option[35], which the webgpu crate is currently also migrating to, named **objc2** [35]. The reason for this are the error-prone manual bindings to the Metal API in the metal-rs crate. Our fourth option, the objc2 crate, provides automatically generated bindings for the Metal API in its framework library **objc2_metal**. Until Apple moved to the programming language Swift, Objective-C was the primary programming language for Apple platforms, and it is still maintained as a native way to interact with the Metal API. "The emphasis on soundness and idiomatic Rust are music to our ears." [33] is how the webgpu crate maintainers describe the objc2 crate. All these reasons led to the decision to use the objc2 crate and its framework crate *objc2_metal* for the implementation of the regex engine on the Apple M1 GPU.

In the previous chapter, we examined various algorithms for regex matching. For our implementation on the Apple M1 GPU, we have selected the Shift-And-Dist algorithm, which offers an optimal balance of simplicity and efficiency. This algorithm supports all operators available in the Rust regex crate, with the exception of loops longer than one character, as the underlying automaton cannot accommodate back-references. Due to the GPU's integer size limitations, our implementation will support regular expressions with a maximum of 64 states, corresponding to the largest integer size available on the M1 GPU. Our implementation objective is to develop a GPU-accelerated regex engine that provides functionality comparable to the Rust regex crate while leveraging the parallel processing capabilities of the M1 GPU within these constraints.

Consequently, we require a constructor that compiles the regex string into an automaton, which can then execute the regex pattern against input strings. In our state structure, we will implement three approaches for string matching: counting matches per line, returning a boolean vector indicating the match status for each line, and counting the number of matches in the input text. These methods are defined as:

- `fn count_matches(&mut self, text: &str)-> usize;`
- `fn count_match_lines(&self, text: &[[u8; BLOCK_SIZE]])-> Result<usize>`
- `fn match_lines(&self, text: &[[u8; BLOCK_SIZE]])-> Result<Vec<bool>>`

As we will see in later Section 4.3.4, the GPU execution can also be done asynchronously, which is why we will also add an asynchronous version of the `match_lines` function, which will return a `Result<impl Future<Output = Vec<bool>>>`.

4.1. Regex Compilation

For our Shift-And-Dist implementation, we need to compile the regex into bit masks. An efficient way to construct these is through an ε-free automata, which can be derived by converting the regex into a non-deterministic finite automaton (NFA) and then a deterministic finite automaton (DFA). In the end, we can still minimize the DFA to reduce the number of states and transitions and finally convert the DFA into a bitmask.

4.1.1. Regex Parsing

Regex parsing is the process of converting a regex string into a regex tree. The regex tree represents the regex string, where each node represents a part of the regex string.

The regex tree is then used to convert the regex string into an NFA.

The rust regex crate provides a regex parser that can be used to parse regex to a *high-level intermediate representation* (HIR) in the crate *regex_syntax*. This abstracts away a lot of the complexity behind regex parsing and allows us to focus on the implementation of the regex engine. The HIR consists of 8 operators, saved in the enum named `HirKind` described in Table 4.1.

Kind	Description
`Empty`	The empty regular expression, which matches everything, including the empty string.
`Literal(Literal)`	A literal string that matches exactly these bytes.
`Class(Class)`	A single character class that matches any of the characters in the class. Can consist of either Unicode scalar values or bytes. May be empty, in which case it matches nothing.
`Look(Look)`	A look-around assertion (e.g., word boundary). Look-around matches always have zero length.
`Repetition(Repetition)`	A repetition operation applied to a sub-expression (e.g., *, +, ?).
`Capture(Capture)`	A capturing group containing a sub-expression.
`Concat(Vec<Hir>)`	A concatenation of expressions. Matches only if each sub-expression matches one after the other. Always contains at least two sub-expressions.
`Alternation(Vec<Hir>)`	An alternation of expressions. Matches if at least one sub-expression matches. Always contains at least two sub-expressions.

Table 4.1.: HirKind enum variants and their descriptions

4.1.2. HIR to NFA

For all the following steps, we create a struct that contains the necessary fields for an automaton. On the struct, we can then implement the methods needed for each step.

Listing 4.1: Automaton struct

```rust
pub type State = usize;

pub struct StateData {
    pub transitions: HashMap<char, BTreeSet<State>>,
    pub is_final: bool,
}

pub struct Automaton {
    pub states: HashMap<State, StateData>,
    pub start: State,
    next_state: State,
    pub start_anchor: bool,
}
```

In Listing 4.1, we can see the struct used for the automaton. We use the predefined type `State` to identify a state. `usize` is the pointer-sized unsigned integer type in Rust, so it can be different depending on the platform the code is running on, but it can be used directly to index vectors or arrays.

The `Automaton` struct contains a hashmap of states, the start state, the next state, and a boolean for whether we have a start anchor ($\wedge$) in the regex. We use a hashmap instead of a vector for the states because the index of the states is not necessarily continuous. Thus, we can have gaps between the indices, especially when we remove ϵ-transitions or minimize the automaton. This would be inefficient to store in a vector.

The `StateData` struct contains the transitions for the state and a boolean for whether the state is accepting. We do not store the accepting states in the `Automaton` struct, In the algorithms, we do not need to know which states are accepting; we only need to know if a state is accepting or not, usually while iterating over each state. To save the transitions, we again use a hashmap. Here, we could have used a vector again since we only use ASCII symbols. However, our vector would be very sparsely populated since a state usually does not have transitions for all 256 ASCII symbols but only for a few. We use a `BTreeSet` to save the destination states for each character. Consequently, we have efficient lookups, as well as sorted iterations over the transitions without any overhead, which is important for the determinization of the automaton.

For the construction of the NFA, we use Thompson's construction algorithm. Consequently, we can have a linear NFA construction runtime in terms of the length of the regex, as the algorithm recursively constructs the NFA from the HIR tree. In each function call, we always match the `HIRKind` and then process the subexpressions accordingly. For example, for the `Literal`, we create a start state and a subsequent state for each character in the literal.

The only case specially handled is the `Look` kind. Since we later on can only handle the basic start (`^a`) and end ($) anchors, we filter for them. If we find a start anchor, we set the property `start_anchor`, and if we add a final transition with the character `0x17`. `0x17` represents the ASCII symbol for the end of transmission (EOT) character, which is not used in the ASCII table and is thus a good choice for an end anchor. We can later append this character to the input and then know when we are at the end of the input.

4.1.3. From NFA to Minimized DFA

Converting a nondeterministic finite automaton (NFA) into a deterministic finite automaton (DFA) is a classical procedure in automata theory that underpins many applications in pattern matching and lexical analysis. The standard method involves two main steps: first, employing the subset construction algorithm to transform the NFA into a DFA and then minimize the resulting DFA using a state reduction algorithm such as Hopcroft's algorithm.

Alternative techniques exist that are more efficient in specific cases. For example, the *follow automaton* construction by Ilie and Yu [17] can perform the conversion in $\mathcal{O}(n(\log n)^2)$ time and yields a number of states bounded by $\frac{3}{2}|n| + \frac{5}{2}$ where n is the length of the regular expression.

However, given that this work focuses on GPU implementation, we opt for the more straightforward subset construction algorithm due to its simplicity and ease of parallelization despite its worst-case time complexity of $O(2^n)$.

The subset construction algorithm works by creating a new state for each subset of states in the NFA. The transitions for the new state are the union of the transitions of the states in the subset. The new state is an accepting state if any of the states in the subset is an accepting state.

Once the DFA is built, we could in principal apply Hopcroft's algorithm to minimize it, partitioning states into equivalence classes and refining them until no further splits are possible. This minimization runs in $O(n \log n)$ time and ensures a more compact and efficient representation of the DFA.

However, due to the underlying structure of the Shift-And-Dist algorithm, Hopcroft's minimization algorithm cannot be effectively applied to DFA in this context. The core limitation stems from the Shift-And-Dist algorithm's inability to handle scenarios where two distinct states transition to the same destination state using different characters. If we look back at Section 3.4.1, we see that this limitation arises because once we calculate the possible next states, we lose information about the originating state, making it impossible to determine which state permitted the transition. Consequently, we cannot reliably identify which character is allowed to transition, rendering the minimization ineffective.

For example, in the regex `(ab|cd)ef`, our constructed automaton will have to transition the characters `b` and `d` to two different states. This is the case because we always calculate all possible transitions for each character and distance and only mask it afterward by the input character. If `b` and `d` would transition to the same state, we would accept the input `adef` or `cbef`. Hopcroft's algorithm would merge the two states, as they have the same next transition (`e`) to the same state from which the final transition with `f` happens. Therefore, we do not apply Hopcroft's algorithm to minimize the DFA.

4.1.4. DFA to GPU

The final step in the regex compilation process involves converting the DFA into a bit mask executable on the GPU. In our implementation, this bit mask is represented by a 64-bit integer where each bit corresponds to a state in the DFA. Since we must align the bits sequentially without any backward edges, we effectively perform a topological sort of the states. Additionally, as our automaton design precludes backward loops, we can employ Kahn's algorithm for topological sorting [24]. This approach provides the added benefit of verifying whether our automaton is acyclic.

Kahn's algorithm functions by initially identifying all nodes without incoming edges and placing them in a queue. Then, while the queue contains elements, we extract a node, add it to the sorted list, remove it from the graph, and examine all its outgoing edges. If a target node of these edges has no remaining incoming edges, we add it to the queue. This process continues until the queue is empty. Should nodes remain in the graph after queue exhaustion, we can conclude that the graph contains a cycle.

Our adaptation of Kahn's algorithm for constructing the bit masks is illustrated in Algorithm 3. For our specific application, we utilize this algorithm to sort the DFA states, beginning with the start state, which we add to the queue. We then iterate through the outgoing edges of each state and incorporate the target states into the queue. During state processing, we add the character and distance masks to the corresponding bit masks(alg. 3 [l. 10–13]), implementing additional logic for initial and final states(alg. 3 [l. 14–17]). Crucially, in cases involving self-loops(alg. 3 [l. 24–26]), we already add the character and distance masks to the bit masks while iterating over the outgoing edges. This modification prevents Kahn's algorithm from failing due to the self-cycles in the graph.

Upon algorithm completion, we verify whether any states remain in the graph. If so, we determine that the automaton contains a cycle and terminate the algorithm. Otherwise, we return the bit masks for characters, distances, and initial and final state masks. This methodical approach ensures that our automaton remains compatible with the efficient bit-parallel algorithms we employ on the GPU while also providing early detection of constructs that could lead to infinite loops or other execution inefficiencies.

Kahn's algorithm offers a time complexity of $\mathcal{O}(V + E)$, where V represents the number of vertices (states), and E represents the number of edges (transitions); the conversion of the topologically sorted DFA to bit masks is a linear-time operation

Algorithm 3 Kahn's Algorithm with Bitmask Construction for a Directed Graph

```
Require: A directed graph $G = (V, E)$ with states labeled by $v \in V$
Require: A distinguished start state $v_s$; a Boolean indicating start anchoring
 1: Initialize a queue $Q$ as empty
 2: Initialize masksChar[0], masksDist[0] for characters $c$ and distances $d$
 3: Initialize maskInitial ← 0, maskFinal ← 0
 4: Compute incomingEdges[$v$] for each $v$ by counting all incoming transitions
 5: $\text{currentDist} \leftarrow \begin{cases} 0 & \text{if start anchoring is true} \\ -1 & \text{otherwise} \end{cases}$
 6: Enqueue $v_s$ into $Q$
 7: incomingEdges[$v_s$] ← 0
 8: while $Q$ is not empty do
 9:     $u \leftarrow Q$.dequeue()
10:     for all edges $(x \to u)$ with character $c$ and distance $d$ do
11:         masksChar[$c$] += $2^{\text{currentDist}}$
12:         if $d \geq 0$ then
13:             masksDist[currentDist − $d$] += $2^d$
14:         else
15:             maskInitial += $2^{\text{currentDist}}$
16:     if $u$ is final then
17:         maskFinal += $2^{\text{currentDist}}$
18:     for all characters $c$ with transitions $(u \to t)$ do
19:         for all targets $t$ of transitions for character $c$ from state $u$ do
20:             if $t \neq u$ then
21:                 incomingEdges($t$) ← incomingEdges($t$) − 1
22:                 if incomingEdges[$t$] = 0 then
23:                     $Q$.enqueue($t$)
24:             else if currentDist ≥ 0 then                ▷ Self-loops
25:                 masksChar[$c$] += $2^{\text{currentDist}}$
26:                 masksDist[0] += $2^{\text{currentDist}}$
27:     currentDist ← currentDist + 1
28:     if currentDist > 64 then
29:         stop and report an error (too many states)
30: return (masksChar, masksDistmaskInitial, maskFinal)
```

proportional to the number of states and edges, contributing negligibly to the overall compilation time, even for complex patterns. This efficiency ensures that the regex compilation phase remains lightweight compared to the pattern-matching execution phase on the GPU, which processes potentially large input streams.

4.2. Shader Code

Now that we have compiled the bit masks, we can proceed with their execution on the GPU. We will follow the workflow described in Section 3.1.3. As for loading in the shader, two approaches are available: precompiling a shader and loading it as a binary or employing *just in time compilation* (JIT) during runtime. Le Glaunec et al., in their HybridSA paper [28], opt for precompiled shaders with bit masks loaded as buffers, as they deploy multiple regexes simultaneously in batches. The JIT approach offers the advantage of hard-coding the automaton bit masks directly into the shader, thereby requiring only the input and output buffers to be passed to the shader without the need to transmit the automaton separately. This reduction in data transfer can yield performance benefits. More importantly, hard-coding the automaton into the shader allows for compiler optimizations specific to the exact pattern being matched.

4.2.1. Shift-And-Dist Shader

Listing 4.2: Shader Code without any optimization

```
kernel void shift_and_dist(device const Automaton& automaton [[buffer(0)]],
                           device uint32_t* output [[buffer(1)]],
                           device const uint8_t* input [[buffer(2)]],
                           uint3 group_id [[threadgroup_position_in_grid]],
                           uint3 local_id [[thread_position_in_threadgroup]]) {
    const uint block_no = group_id.x * THREADGROUP_SIZE + local_id.y * SIMD_SIZE + local_id.x;
    const uint start = block_no * BLOCK_SIZE;
    const uint end = start + BLOCK_SIZE;

    uint64_t states = 0;
    uint matches = 0;
    for (uint i = start; i < end; i++) {
        uint64_t next = automaton.mask_initial;
        for (uint d = 0; d <= automaton.max_dist; d++) {
            next |= (states & automaton.masks_dist[d]) << d;
        }
        char c = input[i];
        states = next & automaton.masks_char[c];
        matches += (states & automaton.mask_final) != 0;
        if (c == 0x17) { break; } // End of input
    }
    output[block_no] = matches;
}
```

In Listing 4.2, we can see the shader code for the Shift-And-Dist algorithm without any optimizations, which can be precompiled and loaded as a binary. We pass the automaton as a buffer, as well as the output buffer and the input buffer. The input buffer contains the input string with each line occupying a `BLOCK_SIZE` bytes; we define `BLOCK_SIZE` as the page size of the M1 Chip which is $2^{14} = 16\,384\,\mathrm{B}$. The output buffer contains the number of matches for each block.

We calculate the block number from the `group_id` (threadgroup index) and `local_id` (thread in threadgroup index). Both are 3D vectors, which indicate the index of the threadgroup and the index of the thread in the threadgroup. In Section 4.3.1, we will discuss how to dispatch the threads and threadgroups to the GPU and how the different sizes impact the performance. We then calculate the current block's start and end index of the input buffer.

The for loop is the same as discussed in Section 3.4.1 and iterates over the input buffer. We then calculate the next state for each character in the input buffer and count the number of matches. The only difference is that we break the loop if we encounter the EOT character. This is a crucial factor because the input buffer is padded with zeros to the next multiple of `BLOCK_SIZE`, and we need to know when we are at the end of the input. Otherwise, we would waste computation time on the padded zeros.

4.2.2. Hard-Coded Masks

A really nice advantage of the Shift-And-Dist algorithm is that the arithmetic body of the `next` mask for loop iteration is only one `bfi` (Bitfield Insert/Shift Left) assembly instruction. Nevertheless, in each loop iteration, we still need to load the distance mask, a 64-bit integer, from the memory. Loading from memory creates a limiting factor as the memory access is slow compared to the computation of the `bfi` instruction. To optimize this, we could either store the distance mask in threadgroup shared memory, which is shared among the whole threadgroup and is faster to access, or hard code the distance mask into the shader.

Hard coding the shader has additional benefits, such as storing the automaton's character masks in the shader together with the start and end masks. Therefore, we reduce the number of memory accesses to only the input buffer. Additionally, we can trim data type sizes of the state and masks to integers with the size of the maximum amount of states. This results in a cut in the computation time since the M1 GPU is more efficient on 32 bit than on 64 bit integers.

Listing 4.3: Shader Code with hard coded masks

```
const uint8_t masks_char[256] = {0, ..., 0, 0x1, 0x2, 0x2, 0, ... 0, };
...
uint8_t state = 0;
for (uint i = start; i < end; i++) {
    uint c = input[i];
    state = (((state & 0x1) << 1) | 1) & masks_char[c];
    if (state & 0x2) { output[block_no] = 1; return; }
    if (c == 0x17) { break; }
}
```

The for loop of the hard-coded shader for the simple regex `a(b|c)` is shown in Listing 4.3. As the corresponding automaton only has three states, we can use `uint8_t` as our state and mask type. We define the masks as a constant array of the same size with 256 elements, one for each element of the extended ASCII table. The distance for the loop is now unrolled into one statement, along with the following state calculation. Furthermore, the initial mask (1 for the start state) and final mask (2 for the final state) are also hard coded into the shader.

We can compare the two shaders in assembly code to further see the benefits of hard coding the masks. Using a tool developed by Dougall Johnson [22], we can extract the assembly code from the compiled shaders. The assembly code for the whole for-loop of the shader without optimization is shown in Appendix A.2. The assembly code for the loop body of the hard-coded shader is shown in Table 4.2. Already, one optimization of the hard-coded shader is that the compiler unrolls the for-loop into 16 loop bodies since we have a `BLOCK_SIZE` that is a multiple of 16. Unrolling is possible because of the small body size of the loop and the fixed size of the loop. The unrolling allows the compiler to optimize the loop body further and reduce the number of jumps and branches in the loop body.

The hard-coded shader only consists of 4 parts. The first part is for loading the input character and the corresponding mask; the second is for updating the state; the third part checks whether the final state is reached, and the last part checks for the EOT character. The whole loop body only has two `device_load`, which are the slowest instructions since they wait for the memory to load the data, while the non-hard coded

Address	Instruction	Description
Initialize loop / Load starting values		
11a:	`device_load 0, i8, x, r5l, r6_r7, 1, signed, lsl 1`	Load input[i] from device into r5l
122:	`wait 0`	Wait for device_load to complete
124:	`device_load 0, i8, x, r4l, u4_u5, r5, signed`	Load masks_char[c] from device using r5 as index
12c:	`wait 0`	Wait for device_load to complete
Main loop body - process one character		
12e:	`iadd r5h.cache, 0, r9l.discard, lsl 1`	r5h ← r9l ≪ 1 (shift state left by 1)
136:	`and r9l.cache, r5h.discard, 2`	r9l ← r5h&0x2 (check bit 1 in shifted state)
13c:	`and r9.cache, r9.discard, r4l`	r9 ← r9&r4l (apply character mask)
142:	`icmpsel seq, r4h, r9.cache, 0, 0, 2`	If r9 == 0 then r4h = 0 else r4h = 2 (output selection)
Check for termination conditions		
14a:	`while_icmp r0l, seq, r9.discard, 0, 2`	Continue loop if r9 = 0 (state & 0x2 == 0)
150:	`jmp_exec_none 0x592`	Jump to end if state & 0x2 is true (success)
156:	`mov_imm r4h, 1`	r4h ← 1 (prepare output[block_no] = 1)
Check for special character condition		
15a:	`while_icmp r0l, nseq, r5l.discard, 23, 2`	Continue if r5l ≠ 23 (0x17)
160:	`mov_imm r5h, 0`	r5h ← 0 (prepare next `device_load`)
164:	`jmp_exec_none 0x592`	Jump to end if c == 0x17 (break)

Table 4.2.: Assembly implementation of character processing loop

shader has the same two plus the additional d times the `device_load` for the distance mask. A difference in the number of assembly instructions needed for handling the bit masks, can also be seen in the handling of 64-bit integers, which are not needed in the hard-coded shader.

Further, the compiler optimizes branching by using the `r4h` register for saving the output value, which is then set to 1 if the final state is reached and 2 if the EOT character is reached. Some additional minor optimizations can be seen as the usage of the `u4_u5` registers for loading of the character mask, which are shared registers among the threads in a threadgroup. And the use of the `.cache` and `.discard` hints for the registers, which are used for caching and discarding of register values.

As we will later see in Section 5.3, the hard-coded shader is still waiting a lot of the time for the memory to load the input buffer and the character mask. The character mask in this example is a very sparse array, with only character symbols in the middle. Accordingly, we can further optimize the shader by adding a check, whether the character is in the ASCII range of the active characters. If it is not, we can skip the state update and directly set the state to 0, without first loading in the character mask. The implementation Listing 4.4 saves us memory access by introducing a small amount of branching, which is not as bad as memory access.

Listing 4.4: Optimized Shader Code with hard coded masks

```
const uint8_t masks_char[256] = {0, ..., 0, 0x1, 0x2, 0x2, 0, ... 0, };
...
for (uint i = 0; i < BLOCK_SIZE; i++) {
    uint c = input[i];
    state = (c < 66 || 101 < c)
             ? 0
             : (((state & 0x3) << 1) | 1) & masks_char[c];
    if (state & 0x4) {output[block_no] = 1; return; }
    if (c == 0x17) { break; }
}
```

4.3. Deploying the Shader

After generating the JIT-compiled shader, we incorporate it into the static portion of the Metal pipeline; this portion is set up only once per shader, while the dynamic elements are configured for each subsequent execution of the shader. The static portion contains the initialization of the following elements:

- **GPU Device**: The device is the physical GPU on which the shader will be executed.
- **Command Queue**: The command queue is the interface for sending commands to the GPU.
- **GPU Library**: The library contains the compiled shader functions. Created by compiling our previously generated shader.
- **Pipeline State**: The pipeline state is the extraction of the shader functions from the library.

The dynamic portion of the Metal pipeline is set up for each execution of the shader. It consists of the following elements:

- **Command Buffer**: The command buffer is the container for the commands that will be sent to the GPU.
- **Compute Command Encoder**: The compute command encoder is the interface for encoding the commands into the command buffer.
- **Input Buffer**: The input buffer contains the input data for the shader.

- **Output Buffer**: The output buffer contains the output data from the shader.

As an input, we expect a pointer to a buffer of ASCII characters as bytes in rows each of a length of `BLOCK_SIZE` bytes `&[[u8; BLOCK_SIZE]];` allows us to not do any post-processing. The input buffer is created by calling `newBufferWithBytesNoCopy` with the storage mode `MTLResourceStorageModeShared` on the `MTLDevice` object. This creates a buffer based on the input buffer without copying the data out of the input array. The input is then a shared object between the CPU and the GPU, taking advantage of the unified memory architecture of the Apple M1 chip.

The output buffer is created using `newBufferWithLength` and for easy reading afterward `MTLResourceStorageModeShared` as well. We allocate the output buffer this way and do not self-allocate it since we do not need the buffer beforehand and can read it back after the shader's execution. Another option would be to already initialize one output buffer static portion of the Metal pipeline. However, since multiple shader executions can happen in parallel, we need to create a new buffer for each execution; otherwise, we will overwrite the output of the previous execution.

4.3.1. Dispatching Threads and Threadgroups

The final stage of a shader pipeline involves configuring the number of threads and threadgroups before committing the pipeline to the GPU. Metal API offers two approaches for this configuration: `dispatchThreads` and `dispatchThreadgroups`. The `dispatchThreadgroups` method allows explicit specification of the 3D vector dimensions for both *threadsPerThreadgroup* and *threadgroupsPerGrid*, creating uniform threadgroups of identical size. In contrast, `dispatchThreads` requires specifying *threadsPerThreadgroup* along with the total desired thread count. The Metal API then automatically generates non-uniform threadgroups along the edges, as illustrated in Figure 4.1. This approach proves advantageous when the input is not evenly divisible

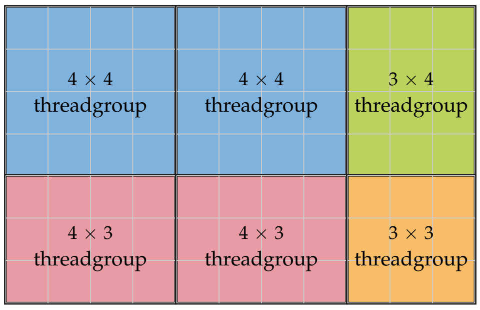


Figure 4.1.: Threadgroups over a 2D-image with nonuniform sizes

by the threadgroup dimensions, as it prevents spawning unnecessary threads that would otherwise remain idle. In the example shown in Figure 4.1, the total thread count is 77, if we would use padding to make the threadgroup size uniform, we would have to spawn $6 \times 4 \times 4 = 96$ threads, resulting in 19 idle threads. Additionally, `dispatchThreads` enables direct retrieval of global grid positions via `uint2 position [[thread_position_in_grid]]`, eliminating the need for manual block position calculations as demonstrated in Listing 4.2.

For optimal performance, Apple recommends configuring the total number of threads per threadgroup (*threadsPerThreadgroup*) to match the maximum allowable value, which can be obtained from the compute pipeline state via `maxTotalThreadsPerThreadgroup`. Specifically for image pipelines, Apple advises using the `threadExecutionWidth` (the SIMD width) for one dimension while allocating the remaining capacity of `maxTotalThreadsPerThreadgroup` to the second dimension. On the Apple M1 GPU, this approach would yield a vector configuration of `(32, 32, 1)`. When deploying a one-dimensional array, Apple's recommendation is to assign `maxTotalThreadsPerThreadgroup` to one dimension while setting the remaining dimensions to one [7, 34].

In Section 5.5, we evaluate and compare the performance of both dispatch methods. When implementing `dispatchThreadgroups`, we follow Apple's recommended approach by setting the first dimension to `threadExecutionWidth` and allocating the remainder to the maximum as the second dimension. The number of threadgroups is calculated by rounding up the quotient of the input array length divided by the threadgroup size. Consequently, this approach may deploy up to `maxTotalThreadsPerThreadgroup` $-$ 1 excess threads, which represents the maximum potential thread overhead. On our hardware, the Apple M1 Chip, this value is 1023. For the second implementation, we configure *threadsPerThreadgroup* to `(maxTotalThreadsPerThreadgroup, 1, 1)` and set the total number of threads to exactly match the length of the input array, resulting in no excess threads.

4.3.2. Indirect Command Buffers

In the previously described approach, we recreate the command buffer from the ground up in each iteration. An alternative is to use a *IndirectCommandBuffer*. Having an indirect command buffer does not keep us from initializing a command buffer each turn but enables us to preserve a pipeline state and buffers for all iterations. In our case, conserving buffers is not really an option since each time has a different input and output buffer. Hence, we only want to save the pipeline state. For this, we create a `MTLIndirectCommandBufferDescriptor` in the static part of the program. With which we specify how many buffers we are going to bind, that want a constant pipeline state, and that we want to inherit the buffers in each iteration from the command buffer. After we initialize the indirect command buffer using that description, we can retrieve the compute command at the first index using `indirectComputeCommandAtIndex(0)`, allowing us to finally set the compute pipeline state. To enable this, we first had to set the compute pipeline state to be indirect.

Ultimately, this multistep process permits us to call `compute_command_encoder.executeCommandsInBuffer_withRange(...)` instead of `compute_command_encoder.setComputePipelineState(...)`. Afterward, we still have to set the thread dispatch sizes, but this time, it is done directly on the indirect compute command. Now, when we commit the command buffer, the already saved pipeline state is used.

4.3.3. Argument Buffers

A final optimization, often mentioned alongside indirect command buffers, are *argument buffers*. These provide an abstraction layer above directly binding input and output buffers to shader functions and command buffers. This abstraction is implemented by creating a struct containing pointers to the buffers that actually hold the content, as shown in Listing 4.5.

Listing 4.5: Argument Buffer struct

```
struct ArgumentBuffer {
    device const uint8_t* input [[id(0)]];
    device uint8_t* output [[id(1)]];
};
```

In the shader preparation pipeline, we create a buffer sized only to accommodate two pointers and populate it with pointers to the input and output buffers. It is crucial to use the GPU addresses of these buffers, retrievable via `buffer.gpuAddress()`, as the GPU operates in its own address space distinct from the CPU. Another essential step is informing the command encoder about the buffers referenced in the argument buffer by calling `compute_command_encoder.useResource_usage(buffer, usage)` for each buffer. Here, we must specify the intended usage: `Read` for the input buffer and `Write` for the output buffer. This specification is vital for the GPU to optimize memory access based on buffer usage patterns. Finally, we bind the argument buffer to the shader function by invoking `compute_command_encoder.setBuffer(argument_buffer, 0, 0)`.

Theoretically, this approach yields performance improvements since the GPU can better optimize memory access patterns and eliminate the need to bind each buffer individually. This optimization is particularly valuable in complex rendering pipelines where numerous buffers operate simultaneously. Furthermore, argument buffers provide a more elegant programming interface, enhancing code maintainability as shader complexity increases.

4.3.4. Output Retrieval

Since we have now deployed the shader and executed it on the GPU, we need to retrieve back the outputs from the GPU. As the execution process on the GPU is asynchronous, we have the option to either block the thread and wait for the execution to finish or add a callback and return a future. In our case, we implement both. The former approach is simpler and just involves calling `command_buffer.waitUntilCompleted()`. The latter is more complex and requires the use of a semaphore to signal the completion of the execution. However, as a benefit, this allows for scheduling more work on the GPU while waiting for the execution to finish. This is especially useful as we can then concurrently encode the next command buffer on the CPU while waiting for the current one to finish, as we will later see in the evaluation in Section 5.4.

Listing 4.6: Future for output retrieval

```rust
struct ShiftAndFutureData {
    result: Option<Vec<bool>>,
    waker: Option<Waker>,
    output_buffer: Option<Id<dyn MTLBuffer>>,
}

struct ShiftAndFuture {
    data: Arc<Mutex<ShiftAndFutureData>>,
}

impl Future for ShiftAndFuture {
    type Output = Vec<bool>;
    fn poll(self: Pin<&mut Self>, cx: &mut std::task::Context<'_>) -> Poll
        <Self::Output> {
        let mut data = self.data.lock().unwrap();

        data.result.take().map_or_else(
            || {
                data.waker = Some(cx.waker().clone());
                Poll::Pending
            },
            Poll::Ready,
        )
    }
}
```

Listing 4.6 shows the simple implementation of the future. We create a struct `ShiftAndFutureData` containing the result, a waker, and the output buffer. The `ShiftAndFuture` struct is then used as a future, which is polled until the result is ready. The result is then returned as a vector of booleans. When creating the future, we give it ownership of the output buffer so that the buffer is not deallocated before the future is resolved.

The handler in Listing 4.7 is bound as our completion handler to the command buffer. It is called when the command buffer is finished executing. In the handler, we lock the shared data, retrieve the output buffer, and copy the result into the shared data. We then wake the waker, which signals the completion of the future. Reading from the output buffer is relatively simple since we have a shared buffer between the CPU and the GPU. That enables us to just take the pointer to the buffer and read the data from it. Since we want to read the data as a boolean vector, we map the bytes to booleans and collect them into a vector.

Listing 4.7: Output retrieval using a future

```
let handler = RcBlock::new(
  move |_command_buffer_ptr: NonNull<ProtocolObject<dyn MTLCommandBuffer
      >>| {
      let mut data = shared_data_clone.lock().unwrap();
      if let Some(output_buffer) = data.output_buffer.as_ref() {
          let output_ptr = output_buffer.contents().as_ptr() as *const u8;
          let output: &[u8] =
              unsafe { std::slice::from_raw_parts(output_ptr, input_len) };
          data.result = Some(output.iter().map(|&c| c > 0).collect());
      }

      if let Some(waker) = data.waker.take() {
          waker.wake();
      }
  },
);
```

4.4. How to Crash a MacBook

All code in this project is written in Rust (except for the shaders), but since we utilize Objective-C bindings for the Metal API and provide raw pointers to the GPU, we occasionally must employ the *unsafe* keyword to circumvent Rust compiler safety checks. This practice introduces numerous opportunities for bugs and system crashes.

During development, I encountered a critical bug where the GPU would crash the entire system when the argument buffer was incorrectly bound to the command pipeline. A properly designed operating system should prevent user-space applications from triggering system-wide crashes, as this violates fundamental security and stability principles. This malfunction likely occurs because the GPU first references an invalid pointer to the argument buffer and subsequently uses an incorrect pointer to the input and output buffers. Such improper memory access causes the GPU to reach memory regions outside its permitted scope, inexplicably freezing the *WindowServer* process responsible for UI rendering. The system watchdog attempts to restart this process, but after two failed attempts within 120 seconds, it initiates a complete system restart, generating the error message: "`panic(cpu 2 caller 0xfffffe001e3299c0): userspace watchdog timeout: no successful checkins from WindowServer (2 induced crashes) in 120 seconds.`" This same problem even occasionally bugs the UI or even causes weird rendering defects in other applications. While this severe bug does not manifest when the implementation is correct, it shows the GPU's considerable power and capacity to compromise system stability when improperly utilized.

5. Evaluation

This chapter presents a comprehensive evaluation of our regex engine implementation on the Apple M1 GPU. Our assessment focuses on three key performance dimensions: input scaling behavior, regex pattern complexity, and comparative performance against the Rust regex engine.

5.1. Setup

All evaluations were conducted on a MacBook Air equipped with the Apple M1 chip, featuring 8 CPU cores (4 Performance and 4 Efficiency cores) and 7 GPU cores, running macOS Sequoia 15.3.1. The system has 16 GB of unified memory. This configuration provides a suitable environment for evaluating the performance of as we have enough memory to store large input data and similar compute power between the CPU and GPU, for a fair comparison.

Our test data consists of randomly generated character sequences with line lengths uniformly distributed between 1 and 16 KiB (the configured page size). This approach ensures our evaluation covers a wide spectrum of realistic input scenarios.

All experiments, unless otherwise specified, were run using the Rust *Criterion.rs* benchmarking library [15]. We configured the library to execute a 3 second warm-up sequences prior to each measurement run, ensuring that initialization overhead and just-in-time optimizations would not influence our performance metrics. In the subsequent sections we will discuss and observe the performance impacts of warm-ups, especially with respect to the caching of input data.

For our experiments, we developed and compared four distinct execution strategies, all utilizing the `dispatchThreads` function for thread management:

- **BasicStrategy**: Implements direct binding of input and output buffers as function arguments, representing the simplest approach.
- **ArgumentBufferStrategy**: Employs an argument buffer abstraction layer for managing input and output buffers, potentially reducing parameter passing overhead.

- **IndirectCommandBufferStrategy**: Utilizes indirect command buffers to preserve pipeline state between invocations, reducing setup costs for repeated operations.
- **CombinedStrategy**: Integrates both the ArgumentBufferStrategy and IndirectCommandBufferStrategy approaches to potentially leverage the advantages of both methods.

The first experiments are all conducted with the **BasicStrategy** to establish a baseline performance. Subsequent evaluations compare the remaining strategies to this baseline.

5.2. Dispatch Commands

Our initial evaluation examines the performance characteristics of two Metal API thread dispatching mechanisms: `dispatchThreadgroups` and `dispatchThreads`. As discussed in Section 4.3.1, these approaches differ in how they allocate computational resources.

The `dispatchThreadgroups` method always dispatches entire threadgroups, potentially resulting in idle threads when the input size is not perfectly divisible by the threadgroup size. In contrast, `dispatchThreads` allocates precisely the required number of threads based on input size.

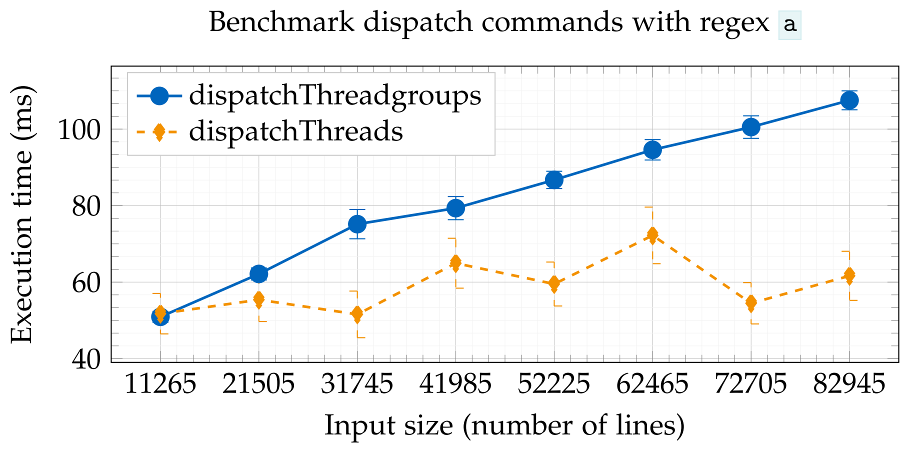


Figure 5.1.: Performance comparison between `dispatchThreadgroups` and `dispatchThreads` commands with the simple regex pattern `a`

Figure 5.1 presents a comparative analysis of these two methods under worst-case conditions, specifically when the number of input lines exceeds the threadgroup size (1024) by exactly one element. To isolate the impact of the dispatching mechanism, we employed a minimally complex regex pattern `a` that would minimize the cost occurred by the regex execution as such, which is not the focus of this evaluation.

The results demonstrate that `dispatchThreads` consistently outperforms `dispatchThreadgroups` across all tested input sizes. While `dispatchThreadgroups` shows linear scaling with respect to input size, `dispatchThreads` does not scale much and maintains an almost constant performance characteristics and consistently achieves lower execution times. This performance advantage can be attributed to the elimination of idle threads and more efficient resource utilization.

These findings clearly indicate that `dispatchThreads` is the superior dispatching mechanism for our regex engine implementation, particularly when processing irregularly sized inputs.

5.3. Just-in-Time Compilation

Next, we analyze the performance impact of just-in-time (JIT) compilation of shader instructions. The compilation time of these instructions is negligibly small compared to the execution time of the regex engine and only occurs once per regex pattern. Our tests indicate that compiling a fresh regex takes approximately 9 ms – about 5 ms for creating the shader library and 4 ms for the actual compilation. macOS provides a system-wide shader cache located at `$(getconf DARWIN_USER_CACHE_DIR)/com.apple.metal`, resulting in compilation times of less than 1 ms for any previously compiled regex patterns. However, during program initialization, the first shader compilation requires slightly less than 10 ms as the cache must be loaded. Since all compilation occurs during the initialization of the shader object rather than during the actual matching phase of the regex engine, and considering that these times are negligible compared to the execution time variances we observe in the regex engine itself, we can safely exclude compilation time from our performance evaluation.

Now let's compare the runtime performance during matching of just-in-time (JIT) vs. pre-compiled shader instructions. In Figure 5.2, we observe the distribution of execution times for JIT-compiled (blue) and pre-compiled (orange) shader instructions. The JIT-compiled shader demonstrates significantly better performance than the pre-compiled shader, with a mean execution time of 224 ms compared to 433 ms—almost half the execution time. Furthermore, the pre-compiled shader exhibits considerably higher variance, while the JIT-compiled shader maintains more consistent execution times.

To understand these performance differences, we utilized Xcode's detailed debugging and performance analysis tools for Metal shaders, which allow precise measurement of executed instructions and instruction-specific execution times. We used these tools to

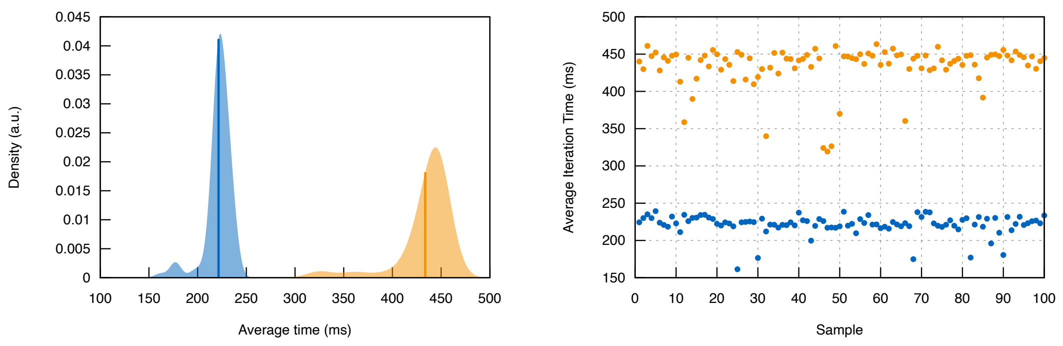


Figure 5.2.: Just-in-time (blue) vs. pre-compiled (orange) distribution of execution times on 1GB of input data with regex `a[^b]{62}b`

compare JIT-compiled and pre-compiled shader instruction performance.

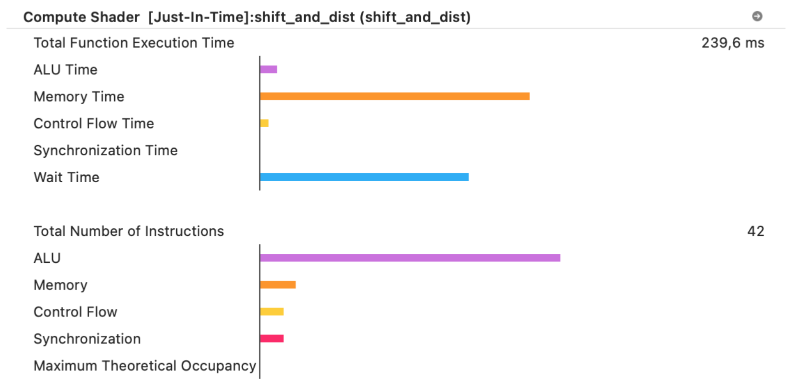
As shown in Figure 5.3, there is a substantial difference between the two compilation approaches. The graph "Total Number of Instructions" reveals that JIT-compiled shader instructions contain approximately half the number of instructions (42) compared to pre-compiled shader instructions (80), though the relative distribution of instruction types remains similar. The graph "Total Function Execution Time" explains this difference: pre-compiled shaders show significantly higher *ALU Time* and *Control Flow Time*. In the JIT-compiled version, many instructions and branching opportunities have been optimized away, resulting in faster execution.

An additional insight from the performance analytics reveals that the pre-compiled shader executes a total of 147,245,300,000 instructions, while the JIT-compiled shader only executes 16,799,500,000 instructions.

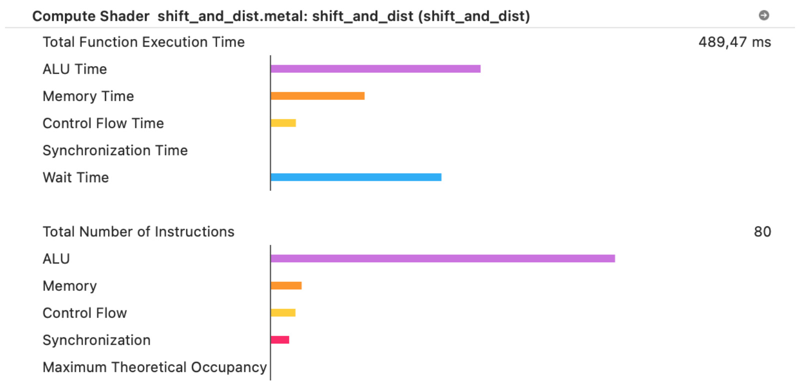
Based on these findings, we conclude that the benefits of JIT-compilation are substantial, with significant performance improvements. Consequently, we will exclusively use JIT-compiled shader instructions in our subsequent evaluations.

5.4. Unified Memory

In much of their marketing material, Apple praises their unified memory architecture (UMA) [4]. Even in their official Worldwide Developers Conference (WWDC) presentations, they state: "Metal exposes the UMA through shared resources that allow the GPU and CPU to read and write the same memory" [19]. Ideally, this should eliminate any need for memory transfers before and after regex matching. However, an important question remains: does this truly eliminate memory latency between the CPU and GPU?



(a) Just-in-time shader



(b) Pre-compiled shader

Figure 5.3.: Comparison of instruction execution times between JIT-compiled and pre-compiled shaders

Using Apple's Instruments tool, we can directly inspect each processing step on the M1 chip. Figure 5.4 illustrates the execution timeline of regex matching with sequential input data, each with a size of 1 GiB. For each execution, we provide only a pointer and the size of the input data to a buffer created using `newBufferWithBytesNoCopy` with the `MTLResourceStorageModeShared` option. For this analysis, we selected a simple `a(b|c)` pattern, as our primary interest lies in measuring memory transfer times.

The Instruments tool provides numerous data points, but only four are relevant for this analysis: *Driver Processing* and its subcategory *Wire Memory*, *Compute*, and *Wired System Memory*. Driver Processing shows GPU driver activities, Wire Memory displays when the driver performs memory wiring, Compute represents the actual GPU computation time, and Wired System Memory indicates the amount of memory

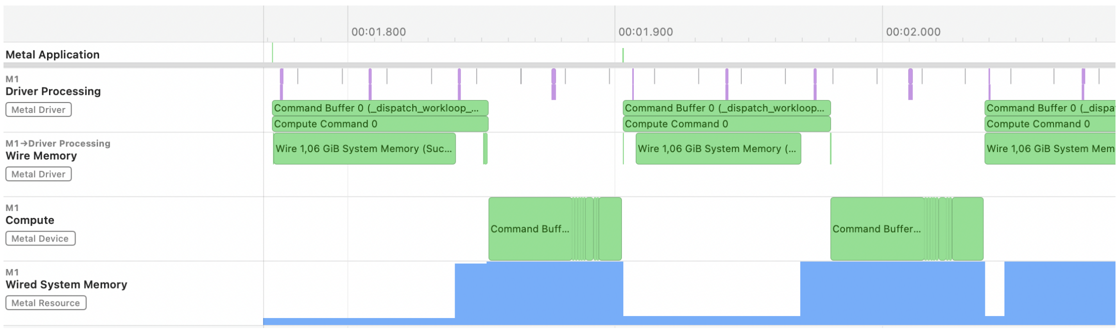


Figure 5.4.: Execution timeline of regex matching with sequential input data

wired to the GPU.

We observe that the driver requires approximately 90 ms to prepare for GPU execution, with about 90% of this time used for wiring 1.06 GiB of memory to the GPU. The actual regex matching requires only about 50 ms. This pattern consistently appears across all subsequent executions.

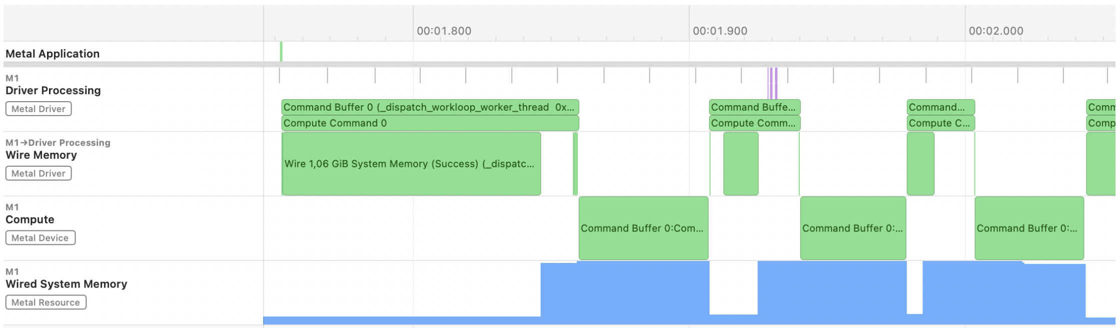


Figure 5.5.: Execution timeline of regex matching with cached input data

When we repeat the same process but use the same input data pointer for all executions, we observe significantly different results. Figure 5.5 shows that all subsequent driver processing times are substantially reduced, with wire memory time becoming almost negligible. This indicates that the memory is already wired to the GPU, eliminating the need for further memory processing. The wiring time drops to less than 10 ms. From these results, we can conclude that while the unified memory architecture requires some initial preparation for GPU execution, once a memory address is wired to the GPU, no additional memory processing are necessary. Further testing reveals that this state persists across multiple program executions. Only after utilizing numerous different memory addresses, likely exhausting some form of cache, does the driver processing time increase again. Since the M1 chip lacks a cache large enough to hold 1 GiB of memory, we hypothesize that the unified memory architecture employs checksums or similar mechanisms to determine whether memory is already wired to the GPU.

Another noteworthy observation is that the *Wired System Memory* metric does not remain elevated after execution. This suggests that memory is not permanently wired to the GPU but only for the duration of execution before being released. This aligns with our observation that the driver performs some wiring operations, even for cached data, before each execution.

Based on these observations, we conclude that the unified memory architecture cannot simply share memory addresses between CPU and GPU without some form of preparation before GPU access. Unfortunately, no documentation is available to confirm what specific operations the driver performs when wiring memory, whether it is copying or only scanning the memory once, we do not know. We can only speculate that the driver might conduct hazard checks or other memory management tasks. However, setting the `HazardTrackingModeUntracked` flag during buffer creation has no impact on wiring time.

The only insight Apple provides regarding CPU – GPU synchronization comes from their WWDC presentation [19]: "Resource management is then about synchronizing the access between the CPU and GPU to happen safely at the right time, rather than duplicating or shadowing data between system memory and video memory." This statement aligns with our observations, as memory is not permanently wired to the GPU but only during execution, and some kind of wiring is happening before execution, which could be part of the synchronization process.

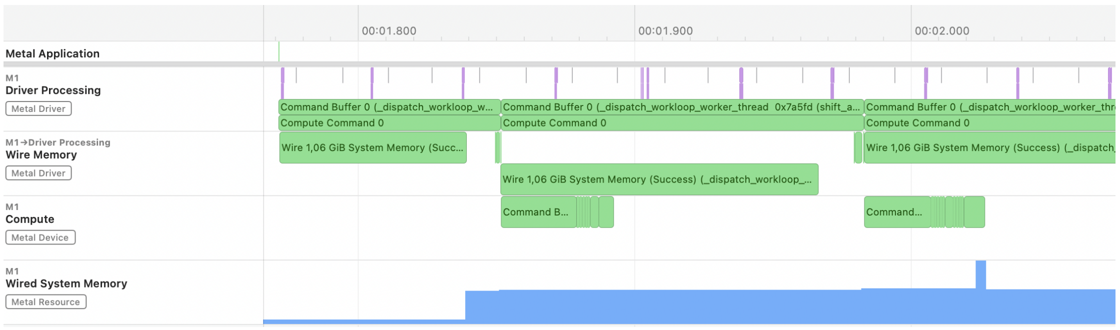

00:01.800
00:01.900
00:02.000
Metal Application
M1
Driver Processing
Metal Driver
Command Buffer 0 (_dispatch_workloop_w...
Compute Command 0
Command Buffer 0 (_dispatch_workloop_worker_thread 0x7a5fd (shift_a...
Compute Command 0
Command Buffer 0 (_dispatch_workloop_worker_thr
Compute Command 0
M1→Driver Processing
Wire Memory
Metal Driver
Wire 1,06 GiB System Memory (Succ...
Wire 1,06 GiB System Memory (Success) (_dispatch_workloop_...
Wire 1,06 GiB System Memory (Success) (_dispatch
M1
Compute
Metal Device
Command B...
Command...
M1
Wired System Memory
Metal Resource

Figure 5.6.: Execution timeline of regex matching with asynchronous dispatching

From these observations, we can infer that asynchronously dispatching regex matching to the GPU could initiate driver processing and wiring operations before the previous execution completes. This approach would address the significant bottleneck presented by driver processing time and wire memory operations, particularly noticeable with small regex patterns. In Section 4.3.4, we presented an approach where we bind a completion handler to the dispatch command instead of waiting for execution completion. This allows us to initiate subsequent executions while the driver is still processing the previous one. Figure 5.6 illustrates the execution timeline of regex matching with asynchronous dispatching. We observe that the driver begins wiring the next memory address to the GPU while the previous execution is still running, thereby mitigating some of the overhead introduced by driver processing and wire memory operations. Additionally, we note a small spike in wired memory at approximately the two-second mark, where two compute buffers have their memory simultaneously wired to the GPU. This spike is absent in the other figures, as they only depict one buffer being wired at a time.

It is worth noting that we observe irregular ripples in the command buffer executions on the GPU. This phenomenon occurs because compute shaders are occasionally interrupted by other executions running simultaneously on the same GPU cores. Since our benchmarks are conducted on macOS, a graphical operating system, display rendering tasks continue to occur during testing, even when minimizing all running applications. As discussed in Section 3.1, Apple GPUs employ a unified compute architecture where the same arithmetic logic units (ALUs) execute both rendering and compute commands. This architectural characteristic, combined with memory caching effects, results in greater variance in GPU execution times compared to CPU executions, as will be discussed in the following section.

5.5. Strategies

Now that we have covered all the basics, let us compare the performance of our four execution strategies alongside a CPU baseline, where the Rust regex engine executes the same regex pattern distributed across 8 threads. We measure 100 samples of the execution time for the asynchronous matching function for each strategy using the aforementioned Criterion.rs library. Although we test the asynchronous matching function, we await the result before initiating the next execution. Since we use the same input data for each execution, we are affected from the memory caching effect discussed in the previous section. Importantly, with the 3 s warm-up period before each measurement, all measured runs already operate with cached input data, and any shader code is preloaded onto the GPU, ensuring our comparisons focus on steady-state performance rather than initialization overhead. Even though cached data does not reflect real world performance scenarios, it allows us to isolate the performance characteristics of the regex engine itself.

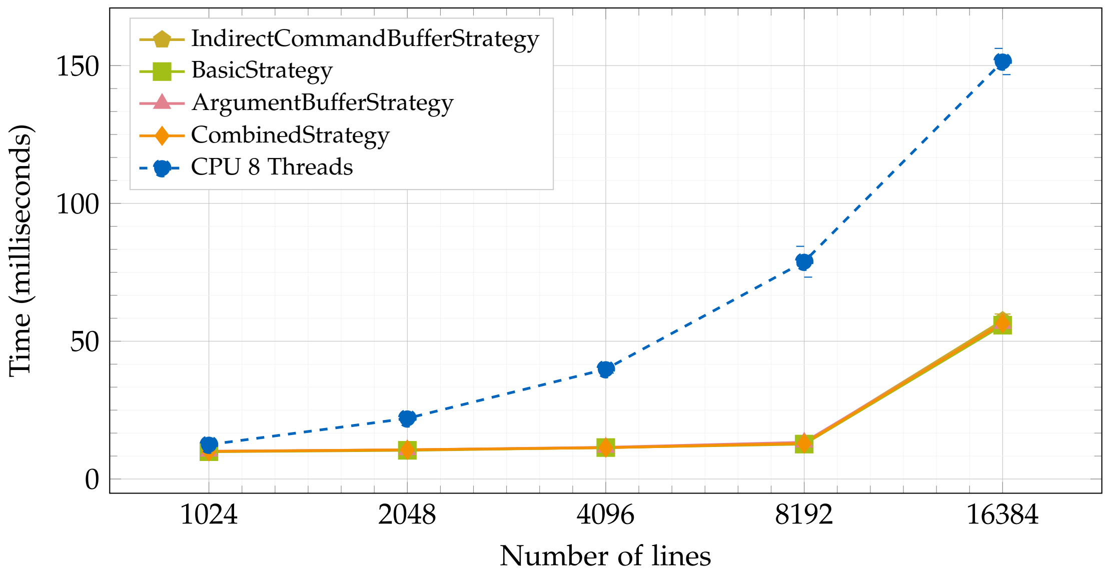


Figure 5.7.: Comparison of execution times of 1024 to 16384 input lines for `a[^b]{62}b`

For our first benchmark, illustrated in Figure 5.7, we evaluate all strategies using between 1024 and 16384 lines of input data with the regex pattern `a[^b]{62}b`. This pattern utilizes all 64 states that can be stored in a `uint64_t` and employs the complete character masks lookup table, thereby simulating the full computational load of the shader without having distance jumps in each iteration of the Shift-And-Dist algorithm. The results demonstrate that the CPU (blue dashed line) is scaling normally with the size of the input data, with execution times increasing linearly. In contrast, the four GPU strategies exhibit remarkable scalability, illustrated by the single line, maintaining constant execution times across the entire range of input sizes from 1024 to 8192 lines. This consistent performance illustrates the substantial parallelization potential inherent in GPU architectures. Furthermore, we observe that all GPU-based strategies perform equivalently, suggesting that the overhead differences between implementation approaches is negligible before sufficient computational load is applied to fully utilize the GPU's processing capabilities.

When extending our analysis to input sizes reaching 1.98 GiB, as illustrated in Figure 5.8, we observe distinct scaling characteristics between implementations. The CPU implementation (indicated by the blue dashed line) exhibits linear scaling with respect to input size, whereas the GPU strategies demonstrate sublinear scaling behavior. This sublinear relationship translates to higher throughput efficiency as input sizes increase. At the maximum tested input size of 1.98 GiB (comprising 262144 input

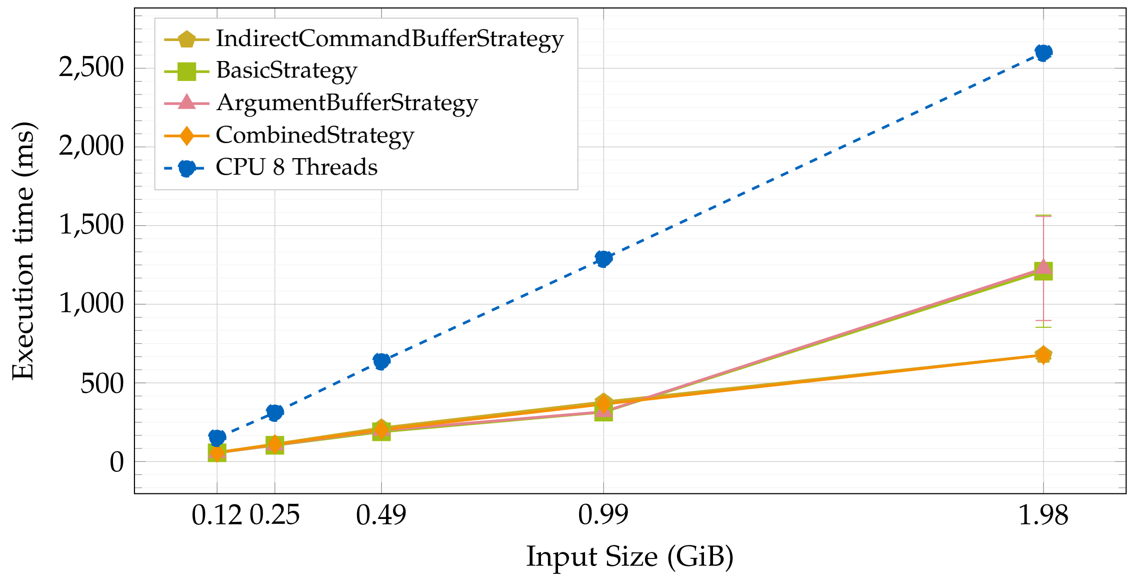


Figure 5.8.: Execution times of the regex `a[^b]62b` on different large input sizes

lines), we again observe general performance parity among most GPU strategies, where the `CombinedStrategy` and `ArgumentBufferStrategy` exhibit significantly degraded performance compared to other strategies, accompanied by substantially increased variance in execution times. As we will explore in detail in Section 5.6, this performance divergence can be attributed to the memory caching behavior associated with processing extremely large input datasets.

By normalizing execution times with respect to input size, we can derive throughput metrics for each implementation strategy, as depicted in Figure 5.9. The CPU implementation (represented by the blue dashed line) demonstrates remarkably consistent throughput of approximately 0.8 GiB/s across the entire spectrum of input sizes. In contrast, the GPU-based strategies exhibit varying throughput characteristics, with the `ArgumentBufferStrategy` and `BasicStrategy` achieving peak performance of 3.16 GiB/s at an input size of 0.99 GiB. This represents a substantial performance advantage, delivering a fourfold speedup relative to the CPU implementation. The remaining GPU strategies demonstrate slightly reduced but still impressive throughput metrics, with the `CombinedStrategy` and `IndirectCommandBufferStrategy` both sustaining approximately 2.70 GiB/s, corresponding to a 3.5-fold performance improvement over the CPU baseline. These results clearly illustrate the substantial throughput advantages that GPU-accelerated regex matching can provide for large-scale text processing tasks.

Overall we can see that the GPU strategies are performing better than the CPU strat-

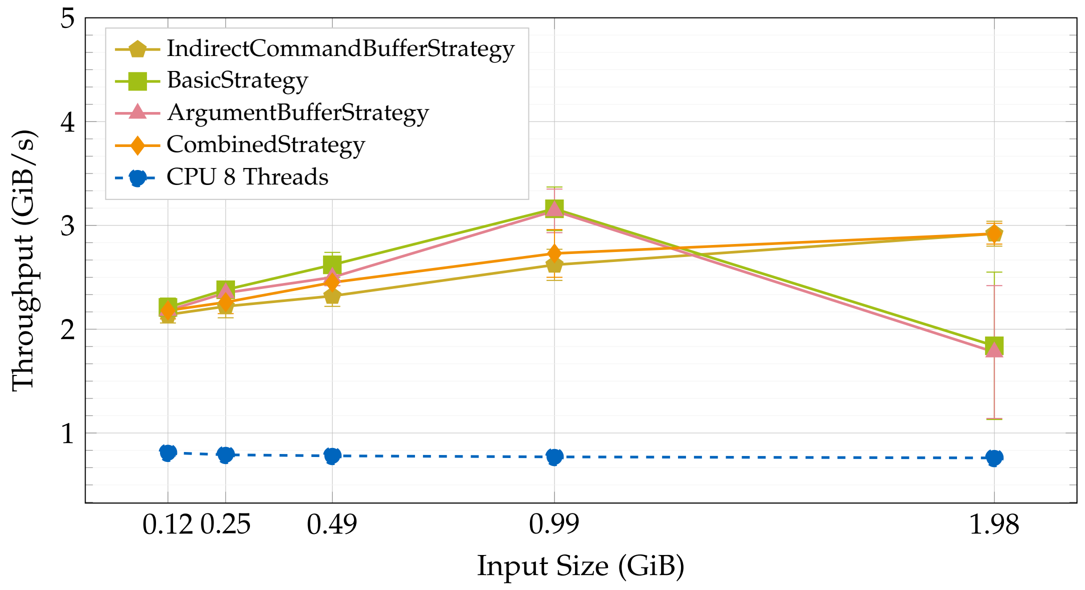


Figure 5.9.: Throughput performance with the regex `a[^b]62b` on different input sizes

egy, with the `ArgumentBufferStrategy` and `BasicStrategy` performing the best, most likely due to their simplicity. However the `CombinedStrategy` and `IndirectCommandBufferStrategy` are only performing slightly worse.

5.5.1. Smaller Regex Patterns

To further explore the performance characteristics of our GPU-based regex matching implementation, we conducted additional benchmarks using smaller regex patterns. In this analysis, we evaluated the throughput performance of each strategy using the regex pattern `a[^b]{30}b` on input sizes ranging from 0.12 GiB to 1.98 GiB. This pattern enables the utilization of a more compact 32-bit state size, thereby reducing the computational load of the shader through more efficient mask lookup operations and the execution of arithmetic instructions using 32-bit rather than 64-bit operations.

As illustrated in Figure 5.10, the throughput characteristics of the GPU strategies maintain consistency with our previous observations, including the performance divergence at the largest input size. The most significant difference appears in CPU performance, which demonstrates substantially higher throughput compared to our earlier benchmarks, now being higher than the GPU implementations before 0.99 GiB input size and then afterwards on parity with the shaders. This improvement stems directly from the reduced computational intensity of the smaller regex pattern. Concurrently,

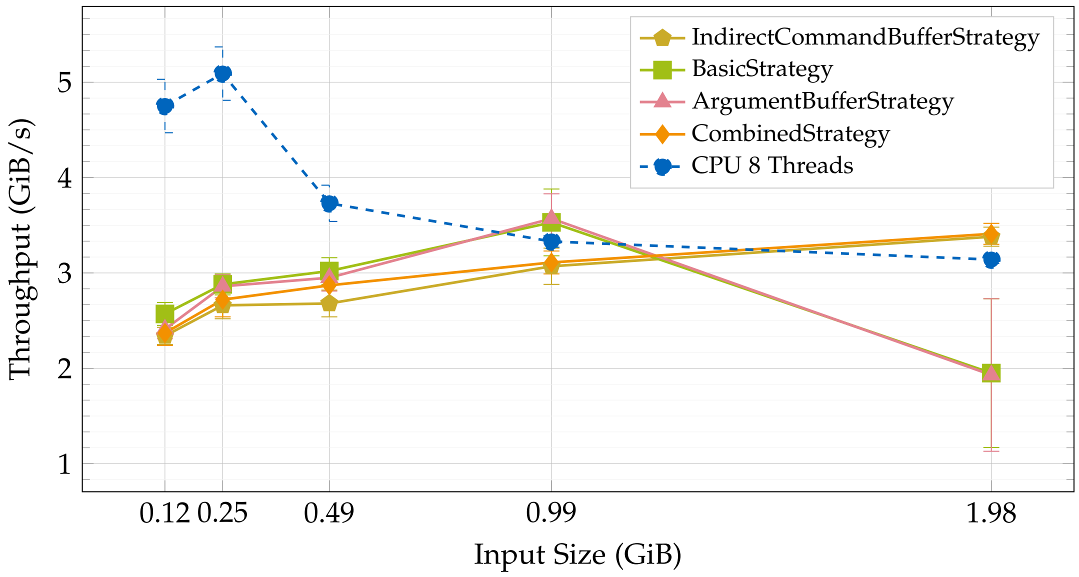


Figure 5.10.: Throughput performance with the regex `a[^b]30b` on different input sizes

the GPU strategies also exhibit enhanced performance metrics, with the `BasicStrategy` and `ArgumentBufferStrategy` achieving peak throughput of 3.57 GiB/s at an input size of 0.99 GiB. The `CombinedStrategy` and `IndirectCommandBufferStrategy` reach 3.07 GiB/s and 3.11 GiB/s respectively, representing an improvement of approximately 0.4 GiB/s compared to our previous benchmarks. Notably, at the maximum input size, these latter strategies marginally outperform the CPU implementation by 0.27 GiB/s.

These results demonstrate that GPU-accelerated regex matching remains competitive with optimized CPU implementations even when processing less computationally intensive patterns. This finding underscores the versatility of our GPU-based approach across diverse text processing scenarios, suggesting that the performance advantages of GPU acceleration extend beyond heavily compute-bound regex operations to encompass a broader spectrum of pattern matching tasks.

5.5.2. Large Jumping Distances

As a third regex we used the regex `a[^b]{0,62}b` which includes all the jumping distances from 0 to 62 characters. Since this regex allows for a lot more matches we expect its throughput to be higher, since we can abort after the first match is found. But additionally we have a lot more computation to do during each input character iteration, as we have a lot more possible next states.

Figure 5.11 illustrates the throughput characteristics of our implementation strate-

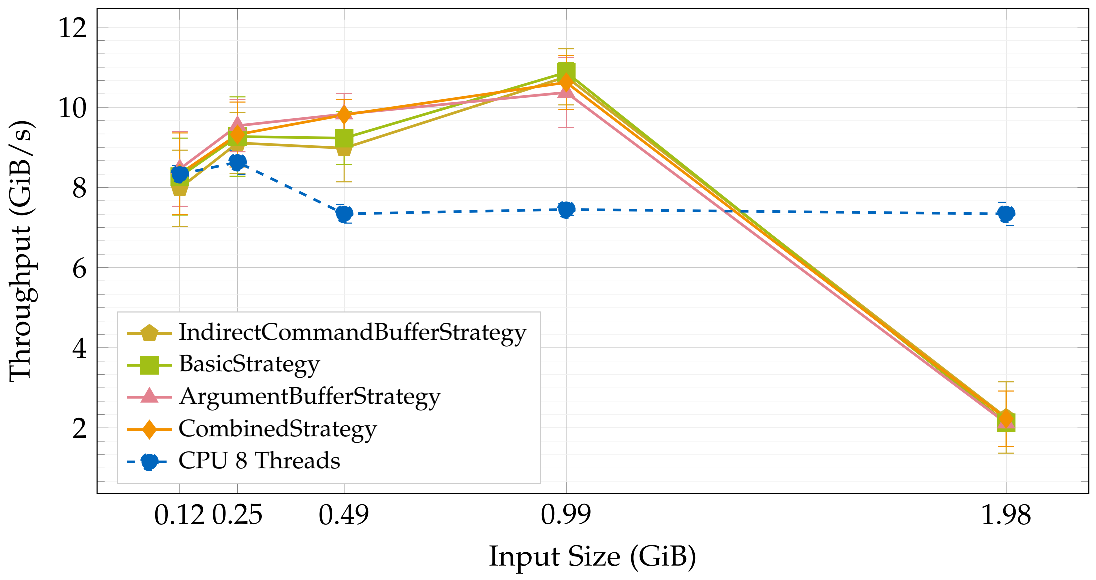


Figure 5.11.: Throughput performance with the regex `a[^b]0,62b` on different input sizes

gies when evaluating the regex pattern `a[^b]{0,62}b` across our standard range of input sizes. As anticipated, this less restrictive pattern yields substantially higher throughput across all implementations. The `BasicStrategy` achieves peak performance of 10.87 GiB/s at an input size of 0.99 GiB, with the alternative GPU-based strategies demonstrating comparable, albeit marginally lower, throughput metrics. The CPU implementation establishes a consistent throughput plateau of approximately 7.40 GiB/s once the input size reaches 0.49 GiB.

A particularly noteworthy observation is the pronounced throughput degradation affecting all GPU strategies at the maximum input size of 1.98 GiB. This performance characteristic aligns with our previous benchmarks and can be attributed primarily to GPU memory caching behavior. This conclusion is supported by the performance distribution of the `CombinedStrategy`, which exhibits several measurement samples achieving throughput equal to or exceeding CPU performance. This bimodal performance distribution strongly suggests that, absent memory caching constraints, GPU implementations would consistently outperform CPU-based approaches even at the largest input sizes. We explore this memory caching phenomenon and its performance implications in greater detail in the following section.

5.6. Chunking

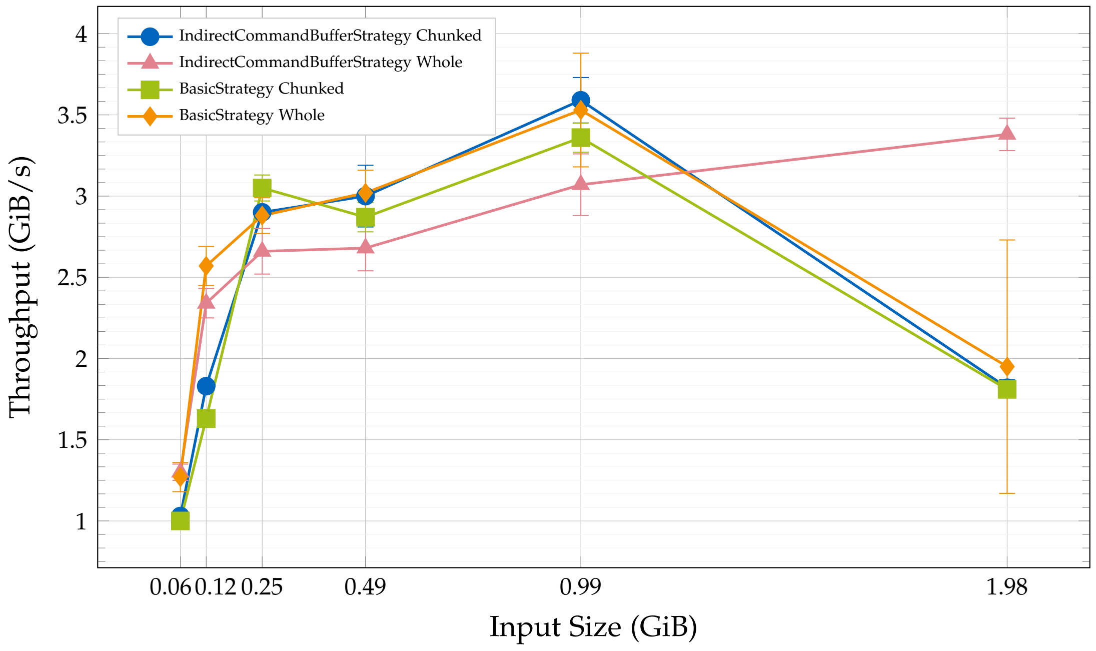


Figure 5.12.: Throughput comparison of chunked and whole strategies with regex `a[^b]{30}b`

Building upon our observations in Section 5.4 regarding the time-saving potential of asynchronous dispatching, we now evaluate the performance impact of chunking input data for GPU processing. As previously noted, this approach allows memory to be wired to the GPU while a prior execution is still in progress. In this section, we systematically compare the performance of chunked versus whole data processing strategies for both the basic and indirect command buffer implementations. Figure 5.12 presents these results, where the input data is consistently divided into four chunks.

Our analysis reveals that the BasicStrategy performs worse with chunking compared to processing the data as a whole. This performance degradation likely stems from the overhead associated with partitioning the input data and managing multiple asynchronous dispatch operations. Conversely, the IndirectCommandBufferStrategy demonstrates improved performance with the chunked approach, as it effectively leverages parallel memory wiring while previous executions continue. This enhanced throughput can be attributed to the IndirectCommandBuffer's preservation of pipeline state, which reduces the overhead when dispatching subsequent chunks. A notable observation is that nearly all strategies exhibit significantly worse performance at 2 GiB, with the exception of the non-chunked IndirectCommandBufferStrategy.

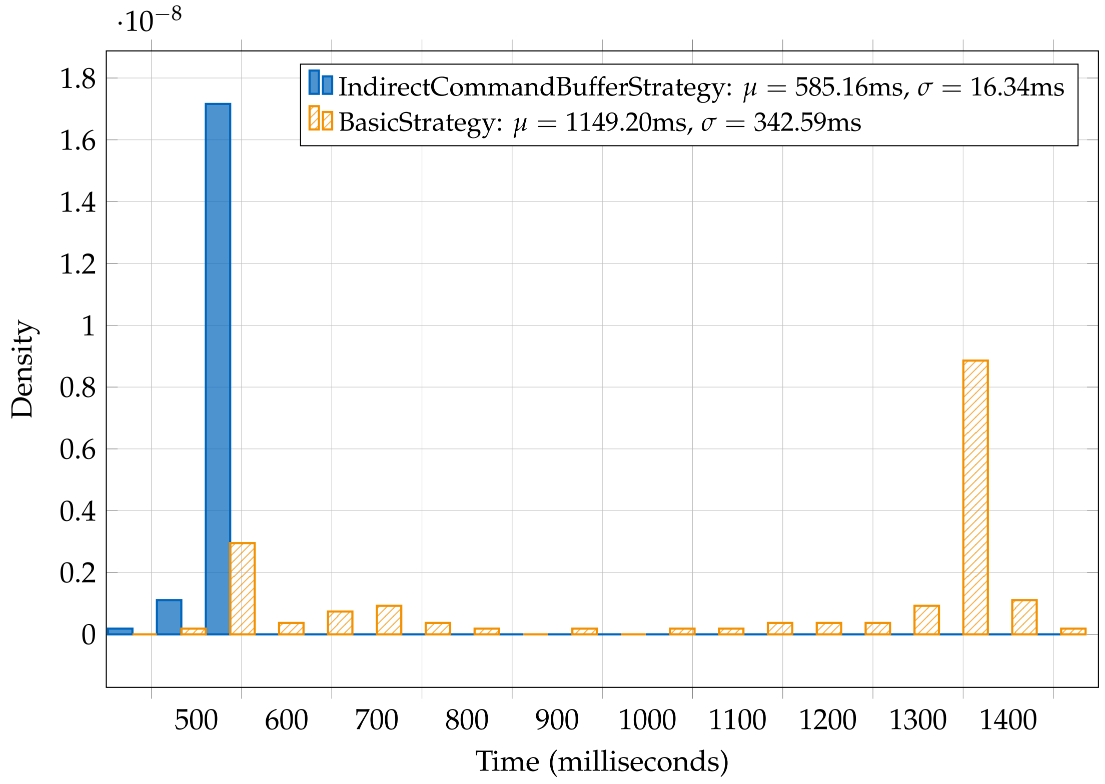


Figure 5.13.: Time distribution comparison between IndirectCommandBufferStrategy and BasicStrategy when not chunked with 1.98 GiB of input data

Further examination of execution time distributions at 1.98 GiB which corresponds to 262144 lines of input data, as illustrated in Figure 5.13, reveals the BasicStrategy exhibits substantially higher variance than the IndirectCommandBufferStrategy, with a distinctive bimodal distribution. These dual peaks correspond to the GPU caching behavior discussed in Section 5.4, suggesting that the IndirectCommandBufferStrategy can still benefit from the caching of the input data. The first peak at about 580 ms in the BasicStrategy aligns with the peak of the IndirectCommandBufferStrategy, indicating that the GPU caching effect is present in both strategies and that the BasicStrategy is equally as fast, when the input data is already cached.

The second peak at approximately 1400 ms in the BasicStrategy distribution corresponds to the memory wiring overhead of approximately 800 ms. While this duration may initially appear excessive, it becomes more reasonable when considering that for each line of input, an entire memory page of 16 KiB must be wired to the GPU. Further investigation indicates that even on the CPU, copying approximately $262144 \times 16\,\text{KiB} = 4\,\text{GiB}$ of data requires approximately 400 ms. Given the opaque nature of the driver's internal operations during memory wiring, we must accept these empirical measurements without the ability to further decompose the contributing factors. This lack of transparency in the driver's memory management processes limits our capacity for more granular performance optimization.

5.7. Summary

Our comprehensive evaluation reveals several key insights regarding GPU-accelerated regex matching on Apple's Metal platform. We established that `dispatchThreads` offers superior performance over `dispatchThreadgroups` by eliminating idle threads and optimizing resource utilization. *Just-in-time compilation* demonstrated substantial benefits, nearly halving execution times compared to *pre-compiled shaders* through more efficient instruction optimization. While Apple's unified memory architecture reduces the need for explicit memory transfers, our analysis uncovered previously undocumented memory wiring operations that significantly impact performance, especially for large inputs. Among our implementation strategies, the *BasicStrategy* and *ArgumentBufferStrategy* consistently delivered the highest throughput, achieving up to a fourfold speedup over CPU implementation for computationally intensive patterns. Performance advantages were maintained even with less complex regex patterns, though the gap narrowed. The impact of chunking varied by strategy, with the *IndirectCommandBufferStrategy* benefiting most from this approach. Memory caching effects emerged as a critical factor at larger input sizes, creating a bimodal performance distribution that reveals both the potential and limitations of GPU-accelerated regex matching. Overall, our findings demonstrate that Metal-based GPU acceleration offers substantial performance improvements for regex matching across diverse text processing scenarios, while highlighting important architectural considerations for optimal implementation.

6. Conclusion

This thesis demonstrated the feasibility and advantages of leveraging Apple's M1 unified memory architecture for high-performance regex execution using GPU acceleration. By exploring the potential of this architecture, we addressed the growing demand for efficient regular expression engines capable of handling large-scale datasets. Our approach bridges the gap between regex theory, GPU programming paradigms, and Apple's hardware innovations developing a regex engine that combines the parallelism of GPU compute shaders with the latency-reducing benefits of unified memory.

Our GPU implementation, built on the *Shift-And-Dist* algorithm and optimized through *Just-In-Time* (JIT) shader compilation, achieves significant performance improvements over traditional CPU-based approaches. Key contributions include a compilation pipeline converting regex patterns into minimized GPU-executable bitmasks, as well as just-in-time shader optimizations to reduce memory latency.

The comprehensive evaluation revealed nuanced performance characteristics across different scenarios:

- The GPU implementation achieved a peak throughput of 10.87 GiB/s for patterns with high amounts of different length transitions like `a[^b]{0,62}b` at 1 GiB input size, outperforming the CPU baseline by 47%. This showcases the GPU's ability to exploit parallelism in computationally intensive regex operations.
- For long regex patterns like `a[^b]{62}b`, the GPU consistently delivered the highest improvement over rust regex library, achieving up to a fourfold speedup over an 8-threaded CPU implementation.
- Even for smaller regex patterns like `a[^b]{30}b`, the GPU delivered 3.57 GiB/s peak throughput, marginally outperforming the CPU by 0.27 GiB/s at 1 GiB of input input data. This illustrates the GPU's versatility across regex complexities.
- For larger inputs (2 GiB), bimodal performance distributions emerged due to memory caching effects. While median GPU throughput remained competitive ($\sim$3.5 GiB/s), some executions suffered from uncached wire memory operations, while cached executions achieved 2.1$\times$ speedups, underscoring the importance of memory-aware scheduling.

- Asynchronous dispatching further reduced latency by overlapping wire memory operations with computation, particularly benefiting chunked processing. Using an *IndirectCommandBuffer* achieved 22% higher throughput when processing 1.98 GiB in four chunks compared to whole-data execution.

JIT-compiled shaders reduced execution times by nearly 50% compared to precompiled variants, primarily by eliminating redundant memory accesses and enabling compiler optimizations tailored to specific regex patterns. The choice of dispatching mechanism also proved critical, with `dispatchThreads` outperforming `dispatchThreadgroups` by minimizing idle threads and optimizing resource utilization.

The scalability of our solution is particularly noteworthy. For inputs exceeding 1 GiB, GPU execution times remained sublinear, showcasing the architecture's ability to exploit parallelism even at scale. However, challenges emerged with larger datasets (2 GiB), where memory caching effects introduced bimodal performance distributions, highlighting the interplay between GPU compute capabilities and memory subsystem limitations.

The results collectively affirm that while unified memory eliminates explicit data transfers, architectural awareness – such as caching behavior, dispatch strategies, and workload chunking – is critical to fully harnessing the M1 GPU's potential. While the unified memory architecture facilitates zero-copy data sharing, our analysis revealed previously undocumented wire memory operations that introduce latency, particularly for large input sizes.

In conclusion, our findings demonstrate that Metal-based GPU acceleration on Apple's unified memory architecture offers substantial performance improvements for regex matching across diverse text processing scenarios. This work provides valuable insights into the architectural considerations for optimal implementation and highlights the potential of leveraging unified memory GPUs for computationally intensive tasks like regular expression matching.

6.1. Future Work

While our experimental results are promising, they have only been tested on synthetic data. Standardized benchmarks such as AutomataZoo [16], which have been used in comparable studies, would provide a more comprehensive evaluation framework. Further evaluation against state-of-the-art approaches presented in the literature is necessary.

We identified some memory limitations of the M1 GPU architecture in our experiments. To address these issues, future work could explore alternative memory management and layout strategies to minimize wire memory instruction delays. For instance, using one pre-allocated buffer into which all input data is copied could reduce the number of memory wiring operations, which could potentially improving performance for larger datasets. Further another layout for the input strings could be used to reduce the number of wire memory operations. Currently, one memory page per input string is a bit excessive and could be reduced to a more compact layout. Or as other research papers have done, interleaving the input strings could be a viable option to additionally improve read performance.

Additional benchmarks across various Apple devices and different chip architectures would be valuable, particularly as the Pro and Ultra series chips offer substantially more graphics computing power than the M1 chip used in our test platform. This would help establish scalability patterns across the Apple Silicon ecosystem. As all Apple devices support Metal shaders even execution on iPads or iPhones is possible allowing for even more potential test platforms.

Since much of the research on the M1 GPU architecture stems from the Asahi Linux team's work, it would be valuable to port our implementation to their drivers, which use OpenCL for shader execution. Given that they have developed custom kernel and user-level drivers for GPU interaction, they might handle the unified memory system differently, potentially offering performance improvements.

Furthermore, integration into production-grade software like DuckDB would provide insight into real-world performance characteristics. DuckDB is a high-performance columnar analytical database system that can run locally as a standalone application with direct API integration. Evaluating how a GPU-accelerated regex engine performs within this context would be particularly informative. In this scenario, benchmarks using TPC-H Query 16 would yield standardized performance metrics for comparison.

Finally, we discussed several Rust GPU libraries at the beginning of this work. It would be worthwhile to determine if similar results could be achieved using the WebGPU crate, which theoretically supports a native objc2 backend and shared GPU buffers. A key question remains whether the overhead introduced by first compiling to the WebGPU Shader Language (WGSL) would preserve comparable performance characteristics.

A. Appendix

A.1. Regex Metacharacters

Character	Description
`.`	Matches any character except a newline.
`[ ]`	Matches any character in the set. `[abc]` or `[a-c]` matches “a”, “b” or “c”
`[^]`	Matches any character not in the set. `[^abc]` or `[^a-c]` matches any character except “a”, “b” or “c”
`^`	Anchors the regex to the start of the string. `^a` matches “a” at the start of the string.
`$`	Anchors the regex to the end of the string. `a$` matches “a” at the end of the string.
`*`	Matches zero or more of the preceding character. `a*` matches “”, “a”, “aa”, “aaa”, etc.
`+`	Matches one or more of the preceding character. `a+` matches “a”, “aa”, “aaa”, etc.
`?`	Matches zero or one of the preceding character. `a?` matches “” or “a”.
`{n}`	Matches exactly n of the preceding character. `a{3}` matches “aaa”.
`{n,}`	Matches n or more of the preceding character. `a{3,}` matches “aaa”, “aaaa”, “aaaaa”, etc.
`{n,m}`	Matches between n and m of the preceding character. `a{3,5}` matches “aaa”, “aaaa”, or “aaaaa”.
`()`	Groups characters together. `(abc)` matches “abc”.
`|`	Alternation, matches either the left or right side. `a|b` matches “a” or “b”.
`\`	Escapes a metacharacter, allowing it to be matched as a literal. `\.` matches “.”.

Table A.1.: Common regex metacharacters and their meanings.

A.2. Shader Assembly Code

Address	Instruction	Description
Initialize states and loop counters		
62:	`mov r5, u8`	Load initial upper `states` value
68:	`mov_imm r8.cache, 0, 0b0`	Initialize `d` loop counter to 0
6e:	`mov r3, u9`	Load initial lower `states` value
Inner loop: for (uint d = 0; d <= max_dist; d++)		
74:	`push_exec r0l, 2`	Start inner loop execution context
7a:	`iadd r10_r11.cache, u0, r8, lsl 3`	Calculate offset: `u0 + d*8`
82:	`iadd r11, r11.discard, u1`	Add base address to offset
8a:	`device_load 0, i32, xy, r10_r11, r10_r11, 131, signed, lsl 1`	Load `automaton.masks_dist[d]`
92:	`wait 0`	Wait for device load completion
Compute: next |= (states & automaton.masks_dist[d]) « d		
94:	`and r12.cache, r8.cache, 63`	`d & 63` (bit position within 64-bit word)
9a:	`and r13.cache, r10.discard, r1`	`states & automaton.masks_dist[d]` (lower part)
a0:	`isub r10.cache, 32, r12.cache`	`32 - (d & 63)` for upper shift
a8:	`bfeil r15.cache, 0, r13.cache, r10.discard`	Prepare upper bits for shift
b0:	`and r14.cache, r11.discard, r2`	`states & automaton.masks_dist[d]` (upper part)
b6:	`bfi r11.cache, 0, r13.cache, r12.cache`	Shift lower bits by `d`
be:	`isub r10.cache, r12.cache, 32`	`(d & 63) - 32` for lower shift
c6:	`bfi r12.cache, r15.discard, r14.discard, r12.discard`	Complete shift operation
ce:	`icmpsel slt, r11.cache, r10.cache, 0, r11.discard, 0`	Handle overflow condition
Update loop counter and accumulate results		
d6:	`iadd r8.cache, 1, r8.discard`	Increment `d`
de:	`or r5, r11.discard, r5`	Accumulate to `next` (upper part)
e4:	`icmpsel ult, r4l.cache, u6, r8, 1, 0`	Compare `d` with `max_dist`
ec:	`bfi r11.cache, 0, r13.discard, r10.cache`	Additional bit field insertion
f4:	`icmpsel ult, r0h.cache, u7, 0, 1, 0`	Control flag setting
fc:	`icmpsel seq, r0h.cache, u7, 0, r4l.discard, r0h.discard`	Loop condition check
104:	`icmpsel slt, r10.cache, r10.discard, 0, r12.discard, r11.discard`	Handle shift edge case
10c:	`icmpsel seq, r0h.cache, r0h.discard, 0, 0, 1`	Additional control flag
114:	`or r3, r10.discard, r3`	Accumulate to `next` (lower part)
11a:	`while_icmp r0l, seq, r0h.discard, 0, 2`	Continue inner loop while condition is true
120:	`jmp_exec_any 0x7A`	Jump back to inner loop start
126:	`pop_exec r0l, 2`	End inner loop execution context
Load and process current character: char c = input[i]		

12c:	`device_load 0, i8, x, r11, u4_u5, r7, unsigned`	Load `input[i]`
134:	`wait 0`	Wait for device load completion
136:	`bfi r1.cache, 0, r1.discard, 24`	Prepare for sign extension
13e:	`asr r1.cache, r1.discard, 24`	Sign extend loaded character
Apply character mask: states = next & automaton.masks_char[c]		
146:	`iadd r1_r2.cache, u0, r1.discard.sx, lsl 3`	Calculate offset for character mask
14e:	`iadd r2, r2.discard, u1`	Add base address to offset
156:	`device_load 0, i32, xy, r11_r12, r1_r2, 3, signed, lsl 1`	Load `automaton.masks_char[c]`
15e:	`wait 0`	Wait for device load completion
160:	`and r1.cache, r11.discard, r5.discard`	`next & automaton.masks_char[c]` (lower)
166:	`and r2.cache, r12.discard, r3.discard`	`next & automaton.masks_char[c]` (upper)
Check for final state: if (states & mask_final)		
16c:	`and r3.cache, r1, u10`	Check against `mask_final` (lower)
172:	`and r5.cache, r2, u11`	Check against `mask_final` (upper)
178:	`or r3.cache, r3.discard, r5.discard`	Combine results
17e:	`icmpsel seq, r0h.cache, r3.discard, 0, 1, 0`	Set flag if match found
186:	`mov_imm r4h, 2`	Prepare return value
18a:	`while_icmp r0l.cache, nseq, r0h.discard, 0, 2`	Check if we should continue
Outer loop increment and condition check		
190:	`iadd r7.cache, 1, r7.discard`	Increment loop counter `i`
198:	`icmpsel ult, r4h, r7.cache, r9.cache, 0, 1`	Compare `i` with `end` value
1a0:	`while_icmp r0l, ult, r7, r9, 2`	Continue main loop while `i < end`
1a6:	`jmp_exec_any 0x62`	Jump back to loop start

Table A.2.: Assembly implementation of automaton state tracking and character processing loop

List of Figures

List of Tables

Listings

Bibliography

[1] *About Metal and This Guide*. URL: https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40014221-CH1-SW1 (visited on 02/23/2025).

[2] A. V. Aho and M. J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search." In: *Commun. ACM* 18.6 (June 1, 1975), pp. 333–340. DOI: 10.1145/360825.360855. URL: https://dl.acm.org/doi/10.1145/360825.360855 (visited on 02/16/2025).

[3] *Apple M1 Chip - Apple*. Nov. 10, 2020. URL: https://web.archive.org/web/20201110184757/https://www.apple.com/mac/m1/ (visited on 02/11/2025).

[4] *Apple Unleashes M1*. Apple Newsroom. URL: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/ (visited on 02/14/2025).

[5] Y. Arafa, A.-H. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz. *Low Overhead Instruction Latency Characterization for NVIDIA GPGPUs*. Sept. 1, 2019. DOI: 10.48550/arXiv.1905.08778. arXiv: 1905.08778 [cs]. URL: http://arxiv.org/abs/1905.08778 (visited on 02/13/2025). Pre-published.

[6] R. Baeza-Yates and G. H. Gonnet. "A New Approach to Text Searching." In: *Commun. ACM* 35.10 (Oct. 1, 1992), pp. 74–82. DOI: 10.1145/135239.135243. URL: https://dl.acm.org/doi/10.1145/135239.135243 (visited on 03/03/2025).

[7] *Calculating Threadgroup and Grid Sizes*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/metal/calculating-threadgroup-and-grid-sizes (visited on 03/08/2025).

[8] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. "iNFAnt: NFA Pattern Matching on GPGPU Devices." In: *SIGCOMM Comput. Commun. Rev.* 40.5 (Oct. 22, 2010), pp. 20–26. DOI: 10.1145/1880153.1880157. URL: https://dl.acm.org/doi/10.1145/1880153.1880157 (visited on 02/15/2025).

[9] *Creating Threads and Threadgroups*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/metal/creating-threads-and-threadgroups (visited on 02/27/2025).

[10] A. Frumusanu. *Apple's M1 Pro, M1 Max SoCs Investigated: New Performance and Efficiency Heights*. URL: `https://www.anandtech.com/show/17024/apple-m1-max-performance-review` (visited on 02/14/2025).

[11] A. Frumusanu. *The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test*. URL: `https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested` (visited on 02/14/2025).

[12] *Gfx-Rs/Metal-Rs*. Rust Graphics Mages, Mar. 4, 2025. URL: `https://github.com/gfx-rs/metal-rs` (visited on 03/05/2025).

[13] *Gfx-Rs/Wgpu*. Rust Graphics Mages, Mar. 5, 2025. URL: `https://github.com/gfx-rs/wgpu` (visited on 03/05/2025).

[14] *Grep(1) - Linux Manual Page*. URL: `https://man7.org/linux/man-pages/man1/grep.1.html` (visited on 02/23/2025).

[15] B. Heisler. *Bheisler/Criterion.Rs*. Mar. 13, 2025. URL: `https://github.com/bheisler/criterion.rs` (visited on 03/13/2025).

[16] T. T. II. *Tjt7a/AutomataZoo*. Mar. 4, 2025. URL: `https://github.com/tjt7a/AutomataZoo` (visited on 03/13/2025).

[17] L. Ilie and S. Yu. "Follow Automata." In: *Information and Computation* 186.1 (Oct. 2003), pp. 140–162. DOI: `10.1016/S0890-5401(03)00090-7`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0890540103000907` (visited on 02/15/2025).

[18] A. Inc. *Harness Apple GPUs with Metal - WWDC20 - Videos*. Apple Developer. URL: `https://developer.apple.com/videos/play/wwdc2020/10602/` (visited on 02/14/2025).

[19] A. Inc. *Metal Compute on MacBook Pro - Tech Talks - Videos*. Apple Developer. URL: `https://developer.apple.com/videos/play/tech-talks/10580/` (visited on 03/13/2025).

[20] N. Jacob and C. Brodley. "Offloading IDS Computation to the GPU." In: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). Dec. 2006, pp. 371–380. DOI: `10.1109/ACSAC.2006.35`. URL: `https://ieeexplore.ieee.org/document/4041182` (visited on 02/16/2025).

[21] D. Johnson. *Apple G13 GPU Architecture Reference*. URL: `https://dougallj.github.io/applegpu/docs.html` (visited on 02/27/2025).

[22] D. Johnson. *Dougallj/Applegpu*. Feb. 15, 2025. URL: `https://github.com/dougallj/applegpu` (visited on 03/07/2025).

[23] D. Johnson. *Firestorm Overview*. URL: https://dougallj.github.io/applecpu/firestorm.html (visited on 02/14/2025).

[24] A. B. Kahn. "Topological Sorting of Large Networks." In: *Commun. ACM* 5.11 (Nov. 1, 1962), pp. 558–562. DOI: 10.1145/368996.369025. URL: https://dl.acm.org/doi/10.1145/368996.369025 (visited on 03/05/2025).

[25] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. "Fast Pattern Matching in Strings." In: *SIAM Journal on Computing* 6.2 (June 1977), pp. 323–350. DOI: 10.1137/0206024. URL: https://epubs.siam.org/doi/10.1137/0206024 (visited on 02/16/2025).

[26] C. Lam. *A Brief Look at Apple's M2 Pro iGPU*. Nov. 30, 2024. URL: https://chipsandcheese.com/p/a-brief-look-at-apples-m2-pro-igpu (visited on 02/14/2025).

[27] C. Lam. *iGPU Cache Setups Compared, Including M1*. Nov. 30, 2024. URL: https://chipsandcheese.com/p/igpu-cache-setups-compared-including-m1 (visited on 02/14/2025).

[28] A. Le Glaunec, L. Kong, and K. Mamouras. "HybridSA: GPU Acceleration of Multi-pattern Regex Matching Using Bit Parallelism." In: *Proc. ACM Program. Lang.* 8 (OOPSLA2 Oct. 8, 2024), 331:1699–331:1728. DOI: 10.1145/3689771. URL: https://dl.acm.org/doi/10.1145/3689771 (visited on 11/17/2024).

[29] H. Liu, S. Pai, and A. Jog. "Asynchronous Automata Processing on GPUs." In: *Proc. ACM Meas. Anal. Comput. Syst.* 7.1 (Mar. 2, 2023), 27:1–27:27. DOI: 10.1145/3579453. URL: https://dl.acm.org/doi/10.1145/3579453 (visited on 12/12/2024).

[30] H. Liu, S. Pai, and A. Jog. "Why GPUs Are Slow at Executing NFAs and How to Make Them Faster." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 13, 2020, pp. 251–265. DOI: 10.1145/3373376.3378471. URL: https://dl.acm.org/doi/10.1145/3373376.3378471 (visited on 03/10/2025).

[31] *Metal Programming Guide*. URL: https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Cmd-Submiss/Cmd-Submiss.html#//apple_ref/doc/uid/TP40014221-CH3-SW1 (visited on 02/27/2025).

[32] "Metal Shading Language Specification." In: (2024). URL: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.

[33] *[Metal] Use 'objc2-Metal' by Madsmtm · Pull Request #5641 · Gfx-Rs/Wgpu*. GitHub. URL: https://github.com/gfx-rs/wgpu/pull/5641 (visited on 03/05/2025).

[34] *Performing Calculations on a GPU*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/metal/performing-calculations-on-a-gpu (visited on 03/08/2025).

[35] *Recommend 'objc2-Metal' Instead of 'metal' · Issue #339 · Gfx-Rs/Metal-Rs*. GitHub. URL: https://github.com/gfx-rs/metal-rs/issues/339 (visited on 03/05/2025).

[36] *Rust-GPU/Rust-Gpu*. Rust GPU, Mar. 4, 2025. URL: https://github.com/Rust-GPU/rust-gpu (visited on 03/05/2025).

[37] *Rust-Lang/Regex*. The Rust Programming Language, Mar. 2, 2025. URL: https://github.com/rust-lang/regex (visited on 03/02/2025).

[38] E. F. O. Sandes and A. C. M. A. de Melo. "CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences." In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. New York, NY, USA: Association for Computing Machinery, Jan. 9, 2010, pp. 137–146. DOI: 10.1145/1693453.1693473. URL: https://dl.acm.org/doi/10.1145/1693453.1693473 (visited on 03/01/2025).

[39] E. A. Sitaridi and K. A. Ross. "GPU-accelerated String Matching for Database Applications." In: *The VLDB Journal* 25.5 (Oct. 1, 2016), pp. 719–740. DOI: 10.1007/s00778-015-0409-y. URL: https://doi.org/10.1007/s00778-015-0409-y (visited on 02/23/2025).

[40] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. "Evaluating GPUs for Network Packet Signature Matching." In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software. Apr. 2009, pp. 175–184. DOI: 10.1109/ISPASS.2009.4919649. URL: https://ieeexplore.ieee.org/document/4919649 (visited on 02/15/2025).

[41] *Tailor Your Apps for Apple GPUs and Tile-Based Deferred Rendering*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/metal/tailor-your-apps-for-apple-gpus-and-tile-based-deferred-rendering (visited on 02/14/2025).

[42] P. Turner. *Philipturner/Metal-Benchmarks*. Feb. 13, 2025. URL: https://github.com/philipturner/metal-benchmarks (visited on 02/13/2025).

[43] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. "Gnort: High Performance Network Intrusion Detection Using Graphics Processors." In: *Recent Advances in Intrusion Detection*. Ed. by R. Lippmann, E. Kirda, and A. Trachtenberg. Berlin, Heidelberg: Springer, 2008, pp. 116–134. DOI: 10.1007/978-3-540-87403-4_7.

[44] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. "Regular Expression Matching on Graphics Hardware for Intrusion Detection." In: *Recent Advances in Intrusion Detection*. Ed. by E. Kirda, S. Jha, and D. Balzarotti. Berlin, Heidelberg: Springer, 2009, pp. 265–283. DOI: `10.1007/978-3-642-04342-0_14`.

[45] S. Wu and U. Manber. "Fast Text Searching: Allowing Errors." In: *Commun. ACM* 35.10 (Oct. 1, 1992), pp. 83–91. DOI: `10.1145/135239.135244`. URL: `https://dl.acm.org/doi/10.1145/135239.135244` (visited on 03/03/2025).

[46] Z. Zhang. "Review on String-Matching Algorithm." In: *SHS Web of Conferences* 144 (2022), p. 03018. DOI: `10.1051/shsconf/202214403018`. URL: `https://www.shs-conferences.org/articles/shsconf/abs/2022/14/shsconf_stehf2022_03018/shsconf_stehf2022_03018.html` (visited on 03/03/2025).

[47] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. "GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching." In: *SIGPLAN Not.* 47.8 (Feb. 25, 2012), pp. 129–140. DOI: `10.1145/2370036.2145833`. URL: `https://dl.acm.org/doi/10.1145/2370036.2145833` (visited on 02/03/2025).