Regex solving using GPGPU with CUDA

Cheng Li, Department of Computer Science

McGill University, Montreal

April, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Abstract

Regular expression (RE) matching is a computationally intensive task that can benefit from modern, high-performance and concurrent computing. There have already been related optimization efforts, such as HyperScan [12], which is based on SIMD instructions for CPUs, and algorithms like iNFAnt [7] and ASyncAP [23] that target GPUs, improving performance by exploiting the mapping between REs and their finite state machine representations. GPU-based RE acceleration methods, however, can suffer from expensive execution costs when an RE has many initial potential state transitions, and performance heavily depends on ensuring algorithm parameters properly match GPU capabilities.

In this thesis, we present a novel study that aims to boost performance and broaden applicability on the GPU side. We introduce a pre-filtering technique that checks the match of simpler RE parts before proceeding to more complex ones. We also optimize the GPU parameters, such as thread occupancy, to avoid naive implementation pitfalls and implement additional optimizations to the state-of-the-art GPU-based algorithm to avoid performance issues caused by edge cases. Our design achieves impressive performance improvement, about 40x faster than iNFAnt and up to 1900x faster than ASyncAP in edge cases, while still maintaining competitive performance in more common cases. The use of our GPU-based optimizations greatly improves the potential for more efficient and versatile RE matching on modern GPUs.

Abrégé

La correspondance d'expressions régulières (RE) est une tâche informatique intensive qui peut bénéficier des capacités modernes de calcul haute performance et concurrentiel. Il y a déjà eu des efforts d'optimisation liés, comme HyperScan [12], qui est basé sur des instructions SIMD pour les CPU, et des algorithmes comme iNFAnt [7] et ASyncAP [23] qui ciblent les GPU, améliorant les performances en exploitant la correspondance entre les RE et leurs représentations en machine à états finis. Cependant, les méthodes d'accélération des RE basées sur les GPU peuvent souffrir de coûts d'exécution élevés lorsque l'expression régulière a de nombreuses transitions d'état potentielles initiales, et les performances dépendent fortement de la correspondance adéquate entre les paramètres de l'algorithme et les capacités du GPU.

Dans cette thèse, nous présentons une étude novatrice visant à améliorer les performances et à élargir l'applicabilité du côté GPU. Nous introduisons une technique de préfiltrage qui vérifie la correspondance de parties de RE plus simples avant de passer à des parties plus complexes. Nous optimisons également les paramètres du GPU, tels que l'occupation des threads, pour éviter les écueils d'implémentation naïve, et implémentons des optimisations supplémentaires à l'algorithme basé sur les GPU de pointe pour éviter les problèmes de performances causés par des cas particuliers. Notre conception obtient une amélioration impressionnante des performances, environ 40 fois plus rapide que iNFAnt, et jusqu'à 1900 fois plus rapide que ASyncAP dans les cas particuliers, tout en maintenant des performances compétitives dans les cas plus courants. L'utilisation de nos optimisations basées sur les GPU améliore grandement le potentiel pour une correspondance de RE plus efficace et polyvalente sur les GPU modernes.

Acknowledgment

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). I extend my deepest gratitude to Professor Clark Verbrugge for his invaluable expertise. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to express my heartfelt thanks to my family. To my parents, thank you for your unwavering support and encouragement throughout my academic journey. Your belief in me has been a constant source of motivation. To my wife, your patience, understanding, and love have been my bedrock. Your support during the challenging times has been indispensable.

Finally, I would like to thank my friends Chen, Huang, and Liu for their camaraderie and encouragement. Your friendship and support have been vital in helping me stay focused and motivated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Fast regular expression(regex, or RE) matching allows for quick text scanning, identifying and extracting relevant patterns, and performing complex searches with minimal delay. This capability is crucial in areas such as DOM search, Javascript application, code search, and network packet inspection (NPI), where speed and precision directly impact user experience and system performance. By optimizing regex matching, systems, and applications can improve interactivity and effectively handle more complex tasks and larger datasets.

Problem domains like network packet inspection have been the target of optimizing REs due to their high throughput requirements. Packets need to be filtered based on their match with a set of regular expressions without slowing down network traffic. For this, previous studies have shown that GPU-based implementations can achieve high performance for RE matching, even though RE matching is inherently sequential [7, 1, 2, 5]. These implementations outperform CPU-based ones by a large margin. However, GPU designs and resources have improved over time, making naive implementations suboptimal and leading to low performance due to failing to take full advantage of GPU capa-

bilities. Moreover, the structure of REs can also affect performance, especially when they contain wildcards or other complex patterns that cause performance bottlenecks.

In this thesis, we propose a novel approach that addresses these challenges. After we discovered that most regular expressions contain one or several static strings, we used a simple pre-filtering technique that allowed us to handle more complex REs without significant performance loss. This makes our approach adaptive and suitable for a broader range of REs. We also optimize our performance by tuning GPU resource usage, such as register allocation, block size, and memory use. We achieve better GPU saturation and higher efficiency. For the prefilter stage, we optimized an existing method and proposed a concurrent, naive method that outperformed the existing one. Through an extensive evaluation, we show that our approach using Snort [35]'s RE dataset obtained remarkable performance improvement, about 40x faster than the classic algorithm iNFAnt [7], and up to 1900x faster than the state-of-art algorithm ASyncAP [23] on a consumer-grade GPU. Although simple in concept, we show that our approach can leverage GPU resources effectively and exploit the structure of REs to achieve orders of magnitude speedup.

In this thesis, we augment the process of RE matching with an optimized pre-filtering stage that eliminates expressive corner cases that otherwise reduce performance. Although pre-filtering has been applied to CPU-based designs, such as Hyperscan and Snort, it is relatively novel in GPU-based designs. Our pre-filtering stage uses a simple algorithm, exploiting the high parallelism available to GPUs. Despite being a naive algorithm, it outperforms more complex designs. We improve the performance of the state-of-the-art ASyncAP algorithm, using profiling information to optimize the use of GPU resources. This adaptation is specific to a family of modern GPU designs, such that the approach could also be applied to future, enduring designs. We conduct extensive experimentation, using multiple RE datasets to evaluate our work. This includes a comparison with a modern CPU-based approach design, as well as the re-implementation of the well-known

iNFAnt and ASyncAP algorithm.

The rest of the thesis is organized as follows. We introduce fundamental background in the next chapter 2. Chapter 3 briefly describes related work on implementing RE matching on GPUs, while chapter 4 discusses details of our algorithms and our optimization strategies. We present our experimental process and results in chapter 5, and the conclusion and future work in the last chapter 6.

Contribution of Authors

Parts of this work are published in The 16th Workshop on General Purpose Processing using GPU (GPGPU 2024) in the paper "Regular Expressions on Modern GPGPUs."

Cheng Li is the primary author of this thesis and is responsible for the research, experimentation, and writing. Professor Clark Verbrugge provided the project's core idea and direction, offering supervisory guidance and content editorial.

Chapter 2

Fundamental background

In this chapter, we will first introduce the concept of SIMD. Then we will briefly introduce the CUDA architecture based on the documentation [28] from NVidia. Then, we will present the compilation and benchmark tools we used for this thesis. Last but not least, we will introduce regular expressions and finite automatons.

2.1 SIMD (Single Instruction, Multiple Data)

SIMD (Single Instruction, Multiple Data) [10] is a parallel computing technique that enables a single instruction to operate simultaneously on multiple data points. In SIMD architectures, a single instruction is broadcasted to multiple processing elements, each of which operates on a different data element. This allows for efficient parallelization of tasks that exhibit data-level parallelism, such as vector and matrix operations.

SIMD is utilized in CPUs to accelerate processing by executing a single instruction across multiple data points simultaneously. CPUs typically feature a few SIMD execution units, such as SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions), which

can process data vectors in parallel. As a result, the SIMD in common desktop CPUs is currently still limited to operating on at most 512 bits of data at a time (e.g., via AVX512). However, GPUs (Graphics Processing Units) specialize in parallel computation and feature numerous SIMD cores organized into massively parallel architectures. While CPUs prioritize versatility and latency-sensitive tasks, GPUs are good at throughput-oriented tasks that can leverage thousands of SIMD cores concurrently. Additionally, GPUs often have more extensive SIMD capabilities, enabling higher parallelism and better performance for parallel workloads.

2.2 GPU Architecture and Parallelism

In this section, we will first introduce the architecture of GPUs, followed by a brief introduction of definitions related to CUDA programming.

The CUDA (Compute Unified Device Architecture) architecture enables the efficient utilization of Graphics Processing Units (GPUs) for graphical and general-purpose computations. Initially developed by NVIDIA, CUDA has become one of the most used parallel computing platforms. This architecture gives us a handy tool to accelerate various computational tasks. Given that GPUs normally have several orders of magnitude more cores and threads than CPUs, we can use CUDA when the computation is computable in parallel.

To better understand the CUDA structure, we will start with an introduction to the structure of GPUs. GPUs are designed with thousands of small, highly specialized cores, each capable of independently executing its set of instructions. This design fosters massive parallelism, allowing GPUs to perform many tasks simultaneously. In contrast to CPUs, which are optimized for sequential processing and control flow, GPUs are designed for data-parallel computations, where the same operation is applied to a large dataset. This

makes it possible to enable developers to design appropriate algorithms, divide tasks into parallel threads, and launch them on the GPU to get a better performance. Following are some core architecture and concepts of GPUs:

Streaming Multiprocessors (SMs)

Modern GPUs consist of multiple Streaming Multiprocessors (SMs), which are individual processing units on the GPU chip. Each SM contains a set of CUDA cores, load and store units, special function units(SFUs), and memory caches. SMs are responsible for executing the actual parallel processing tasks. The number of SMs for each GPU depends on the GPU model. Different SMs do not necessarily operate in synchrony with each other. Each SM can execute its own set of warps independently, which aligns with a MIMD (Multiple Instruction, Multiple Data) architecture. This means that different SMs can execute different instructions on different data sets simultaneously. The detailed structure of a steaming multiprocessor is shown in Figure 2.1.

CUDA Cores

Within each SM are thousands of CUDA cores, also known as shaders or processing cores. CUDA cores are responsible for executing individual instructions in parallel. They are highly specialized processors within a GPU that handle parallel computing tasks

Threads and Warps

In the context of CUDA, threads are the smallest units of work that can be scheduled on the GPU. Thousands of threads can run concurrently on a GPU, taking advantage of the massive parallelism inherent in the architecture. Different threads can be synchronized by calling synchronization methods, like __syncthread().

Warps are groups of 32 threads that execute in parallel on an SM. In NVIDIA GPUs, a warp

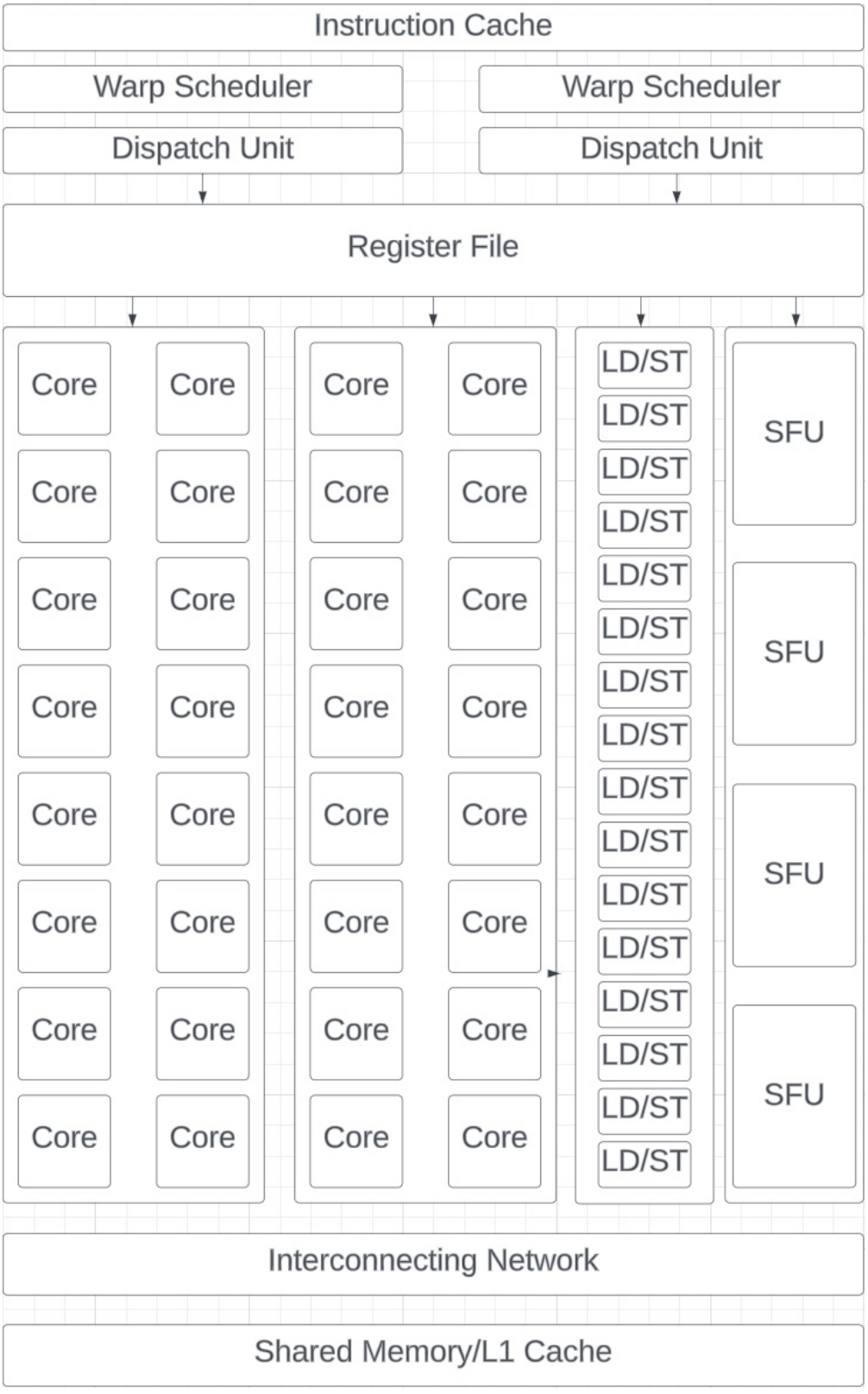


Figure 2.1: Illustration of a Streaming Multiprocessor

is the basic unit of execution. The SM executes instructions on warps, which are scheduled based on available resources. The lock-step execution occurs at the warp level. When a warp is scheduled for execution, all threads in that warp execute the same instruction simultaneously, which is the essence of SIMD.

.

Warp Schedulers

Warp schedulers are responsible for selecting which warp to execute on the SM at any given time. They prioritize ready warps and switch between them to maximize GPU utilization. Schedulers help hide memory latency and keep execution units busy. The architecture after Kepler (GeForce 600 series) features four warp scheduler units per SM.

Occupancy

Occupancy in CUDA refers to the ratio of active warps to the maximum number of warps that can be simultaneously executed on a GPU's streaming multiprocessors (SMs). It essentially measures how effectively the GPU's resources are utilized. Higher occupancy indicates better utilization of the GPU's compute resources, leading to potentially higher performance.

Thread Blocks

Threads are also organized into groups called thread blocks. Thread blocks provide a way to group related threads that can cooperate and synchronize within the block. They are scheduled to run on SMs, and multiple thread blocks can run simultaneously. Each block is composed of multiple warps, and the scheduler on the GPU manages the execution of warps on the available SMs.

The number of threads per block is a key variable in experiments because it directly influences occupancy. The selection of an optimal block size is crucial for maximizing performance, as it balances the trade-offs between shared memory usage, occupancy, and the number of threads that can be executed concurrently. We will discuss how to determine the number of threads per block(block size) later in Section 5.2.

Memory Hierarchy

GPUs have different types of memory: registers, local memory, shared memory, constant memory, texture memory, and global memory. First, each thread has its own set of registers for storing data. Registers are fast memory, and efficient code optimizes the use of these registers to maximize performance. Each thread can also use local memory for thread-specific data. However, access to local memory is slower than registers. Shared memory is a fast, on-chip memory that can be used for data that needs to be shared among threads within the same thread block. Efficient use of shared memory can significantly improve performance. Constant Memory is a read-only memory that is shared among all threads. It is suitable for storing constant values. Texture Memory is optimized for 2D and 3D texture accesses, commonly used in graphics operations. Global Memory is the main memory space for the GPU. It is slower than registers but larger and shared across all threads. The detailed memory hierarchy is shown in Figure 2.2.

GPU memory is separate from CPU memory, and thus, data needs to be transferred between the two. Data transfer from CPU memory to GPU memory involves several steps. First, the CPU prepares the data to be transferred by allocating memory and organizing it into appropriate data structures. Then, the data is transferred from the CPU's main memory (RAM) to the GPU's global memory using specialized data transfer mechanisms such as PCI Express (PCIe) buses or NVLink interconnects. This transfer process typically involves DMA (Direct Memory Access) operations, where the CPU instructs the system's

memory controller to move data directly between the CPU and GPU memory without involving the CPU itself. Once the data reaches the GPU memory, it can be accessed by the GPU for processing, such as in parallel computations or rendering tasks. Data transfer from the GPU to the CPU typically uses APIs such as CUDA's cudaMemcpy with the cudaMemcpyDeviceToHost parameter. The transfer can be synchronous or asynchronous, with direct memory access (DMA) often facilitating faster transfers by allowing the data to move without heavy CPU involvement.

While GPUs offer significant advantages for parallel processing, the memory transfer protocol and associated overhead can pose challenges for high data rate applications. Careful optimization and design strategies are essential to ensure that the benefits of GPU acceleration are realized without being affected by memory bottlenecks.

Parallelism and Data-Parallel Computations

GPUs are optimized for data-parallel computations, where the same operation is applied to a large dataset. This massive parallelism allows developers to divide tasks into parallel threads and execute them simultaneously, which leads to significant speedups in various applications, from scientific simulations to deep learning.

Understanding the architecture and the concept of parallelism is crucial for writing efficient CUDA programs. Developers need to optimize their code to use the various memory types efficiently, minimize memory transfers between the CPU and GPU, and fully take advantage of the GPU's parallel processing capabilities. This can result in substantial performance improvements for various applications when done correctly.

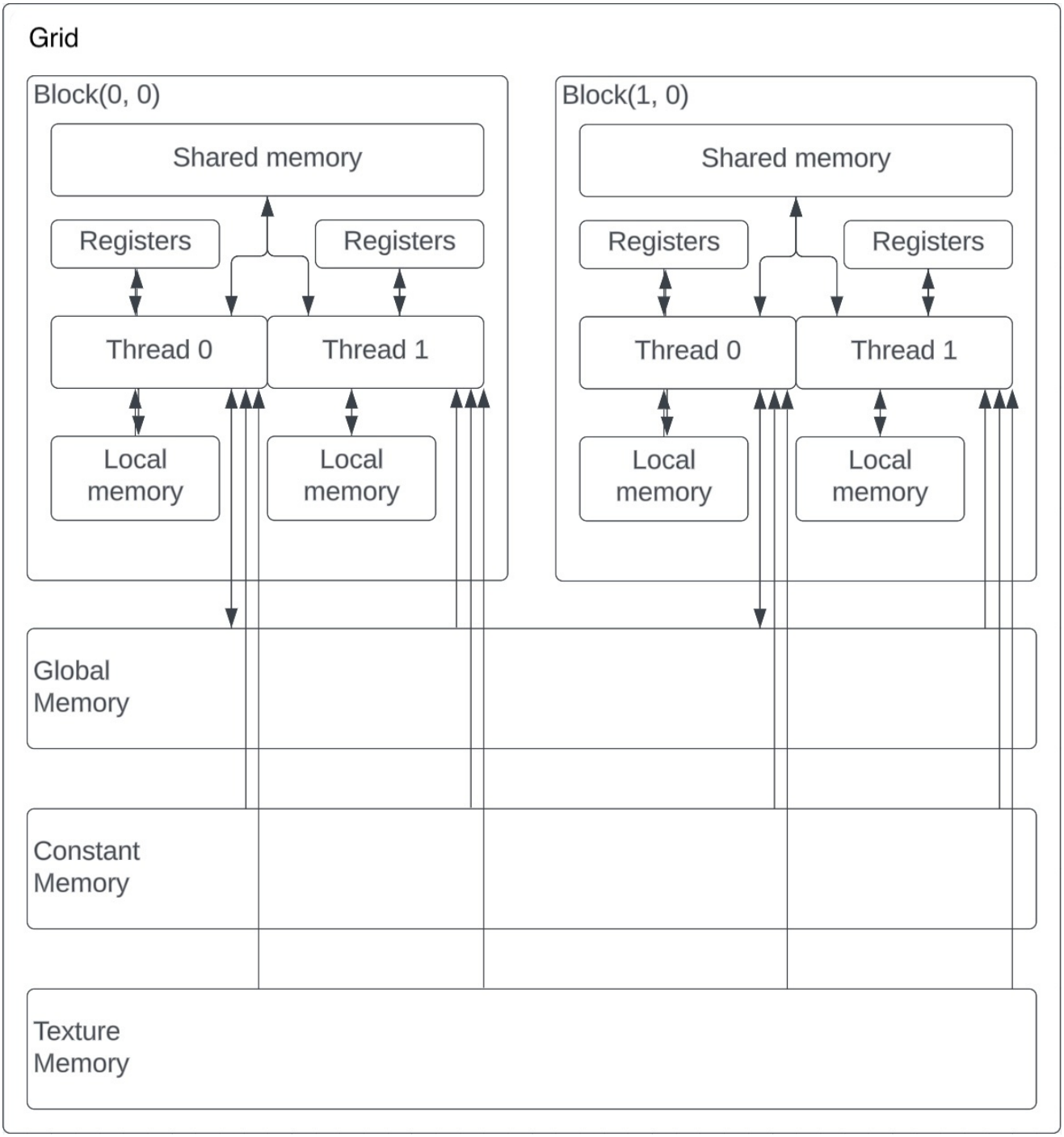


Figure 2.2: Illustration of the memory hierarchy CUDA devices

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Figure 2.3: CUDA code example

2.3 GPU compilation and Tools

We use the CUDA Compiler Driver NVCC 12.3 [27] provided by NVidia to compile our project. At the same time, we use NSight Systems and NSight Compute [26], also provided by NVidia, to perform performance testing.

2.3.1 NVCC

When compiling CUDA code with NVCC, the process begins with preprocessing, parsing, and segregating the code into host and device sections. The host compiler compiles host code, usually standard C or C++, into object files. In contrast, NVCC compiles device code into PTX intermediate representation, which is placed in a "fatbinary." Then, both sections are linked together, with NVCC managing GPU runtime libraries and linker options. Lastly, the host's compiler takes this adjusted program with the "fatbinary" and turns it into a host executable. The compilation steps are shown in Figure 2.4. Figure 2.3 shows a simple CUDA file, where the main function is the host section and the cuda_hello function is the device section. The identifier "__global__" indicates that the following function will run on GPUs. The numbers within the triple angle brackets configure the number of blocks and block size, respectively.

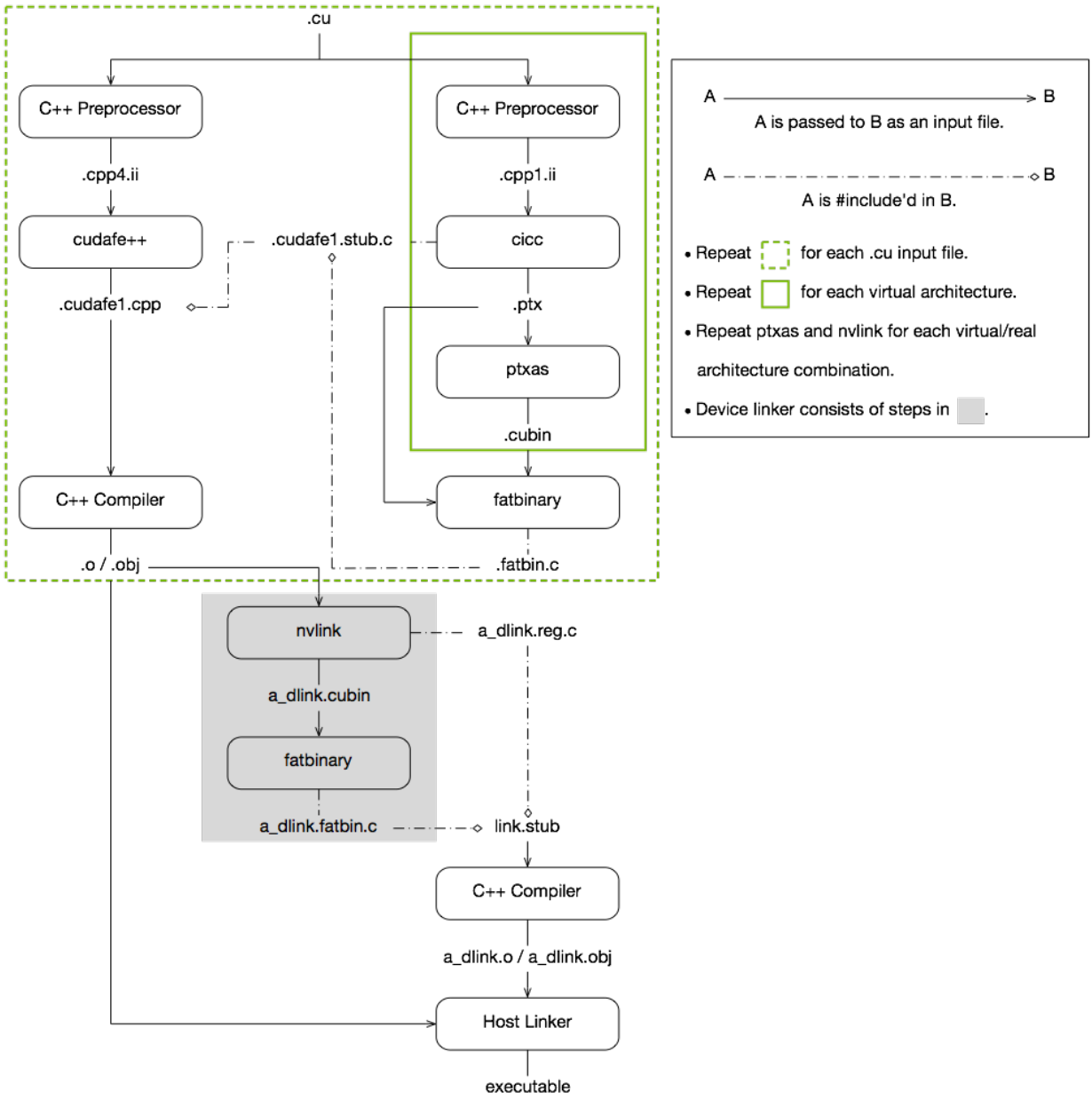


Figure 2.4: Steps of NVCC compilation [27]

2.3.2 NSightSystems and NSightCompute

Given the complexity of the GPU, it is important to get the performance details. NVidia provided us with its benchmark tools, NSightSystems and NSightCompute. However, they have different benchmark scopes. NSightSystems is focused on a higher application level, while NSightCompute is for more detailed information within a device function. The workflow first uses NSightSystems to check the code overall to see if there are any problems. Then, NSightCompute was used to identify the problems in the functions.

A general workflow for NSightSystems and NSightCompute is as follows:

Profiling Configuration: In both tools, users configure the profiling session by specifying the CUDA application they want to analyze and any relevant parameters, such as the CUDA device to use.

Profiling Execution: After configuring the profiling session, users run the target CUDA application from within the client application.

Data Collection: During application execution, both Nsight Compute and Nsight Systems collect performance data, including metrics and timeline information.

Analysis and Visualization: Once the profiling session is completed, users analyze the collected data using the tools' built-in analysis and visualization features. NSightCompute focuses on detailed metrics specific to GPU kernel execution, while NSightSystems provides a broader view of CPU and GPU activities.

Identifying Optimization Opportunities: Based on the analysis results, users identify optimization opportunities, performance bottlenecks, and areas for improvement in their CUDA application code.

2.4 Regular Expression and Finite Automatons

A regular language is a type of formal language in theoretical computer science that can be described by a regular expression or recognized by a finite automaton. It consists of strings formed from a finite alphabet, following certain rules or patterns. Due to their simplicity and well-defined mathematical properties, regular languages hold importance in various computational tasks, such as lexical analysis in compilers, string-matching algorithms, and text-processing applications. A regular language is typically described by a regular expression. Regular expressions are symbolic representations of patterns in strings. They are used to describe sets of strings according to certain rules. Regular expressions typically include a minimal set of operators that can be combined to form patterns for matching strings. They can also be used as patterns to find if they belong to a string.

A basic set of RE operators consists of concatenation, alternation, repetition, and grouping. Concatenation represents the combination of two regular expressions. If r1 and r2 are regular expressions, then their concatenation r1r2 matches any string that can be formed by concatenating a string matched by r1 followed by a string matched by r2. Alternation is represented by the | symbol. If r1 and r2 are regular expressions, then r1|r2 matches any string that is matched by either r1 or r2. The Kleene star * is a unary operator used for repetition. If r is a regular expression, then r* matches zero or more occurrences of the regular expression r. Operator precedence is not always clean, and thus, RE often uses parentheses, which are used to group parts of regular expressions together, similar to how they are used in arithmetic expressions.

Regular expressions can also be represented in a BNF (Backus-Naur Form) grammar, another notation for describing nested language constructs. Here is a simple BNF definition of regular expressions incorporating the four fundamental operations: concatenation, alternation, Kleene closure, and grouping:

<regex> ::= <term> | <term> "|" <regex>

<term> ::= <factor> | <factor> <term>

<factor> ::= <primary> | <primary> "*"

<primary> ::= <char> | "(" <regex> ")"

where <regex> represents the entire regular expression. <term> represents a sequence of factors possibly separated by concatenation. <factor> represents a primary expression possibly followed by the Kleene closure operator. <primary> represents either a single character <char> or a grouped regular expression. <char> represents a single character from the alphabet.

Regular expressions are widely used for pattern matching and parsing in various applications, such as text processing, data validation, and lexical analysis in compilers. When applied to pattern matching and parsing, regular expressions search for specific patterns within input strings. One important concept in regex applications for pattern matching is the idea of "longest match" or "greedy matching." When a regular expression is applied to an input string, it attempts to find the longest substring that matches the pattern specified by the regular expression.

We also have syntactic sugar extensions in regular expressions that provide shorthand notations for common patterns and operations, enhancing readability and writability. Examples include \d for any digit, \w for any word character, and \s for any whitespace character. Quantifiers like +, *, and ? match one or more, zero or more, or zero or one occurrences of the preceding element, respectively.

Another modern extension of regular expressions is the Perl-compatible regular expression (PCRE) [32], which supports more operators and features, such as lookahead, lookbehind, atomic group, conditional, and recursion. Lookahead and lookbehind in PCRE

regular expressions overcome the limitations of regular languages by allowing patterns to be matched based on the context of characters surrounding the current position in the input string without consuming those characters. This capability enables regex patterns to express dependencies and conditions that extend beyond the simple concatenation, alternation, and repetition found in regular languages. PCRE also allows different flags to modify the behavior of the regular expression, such as case-insensitive, multi-line, or ungreedy. PCRE is widely used in programming languages and tools that work with text. For example, the PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl. A full, formal BNF definition of Perl-style regular expression also exists [6].

Finite automatons and regular expressions are dual representations of regular languages. Whereas regular expressions are concise patterns that describe strings in a language, finite automatons are state machines that transition between states based on input symbols. Finite automatons are pivotal in various areas of computer science, such as formal language theory, compiler design, and text processing, and provide an efficient way of matching regular expressions. A finite automaton consists of a set of states, an input alphabet, a transition function, an initial state, and a set of accepting states. The automaton processes input strings by transitioning between states in response to characters from the input alphabet. Two prominent classes of finite automaton are the Deterministic Finite Automaton (DFA) and the Non-deterministic Finite Automaton (NFA).

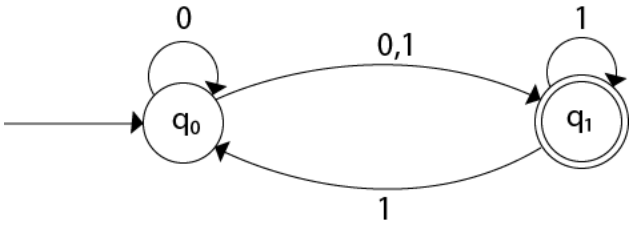
2.4.1 Deterministic finite automaton

A Deterministic Finite Automaton (DFA) is a specific type of Finite Automaton characterized by its deterministic behavior. In a DFA, each state and input character has precisely one defined transition leading to another state. This determinism means that given a specific input string, a DFA will follow a unique path through its states, ultimately accepting

or rejecting the input. DFA execution is highly efficient and predictable, making it well-suited for applications where input patterns have a strict, unambiguous structure. However, the key drawback of DFAs is their inability to handle non-determinism, making them less amenable to parallelization and less suitable for handling complex patterns.

2.4.2 Nondeterministic finite automaton

In contrast to DFAs, Non-deterministic Finite automatons (NFAs) exhibit non-determinism in their state transitions. An NFA can have multiple transitions for the same state and input symbol, offering multiple possible paths for a given input string through the automaton. This non-deterministic behavior makes NFAs more expressive and adaptable when dealing with complex, ambiguous, or irregular patterns. NFAs are especially useful in regular expression matching, where patterns may include optional components, repetition, or alternatives. While NFAs offer greater expressive power, their non-deterministic nature can result in multiple possible outcomes for a given input, making the outcome less obvious and more challenging for sequential processing. Figure 2.5 shows an equivalent pair of NFA and DFA.



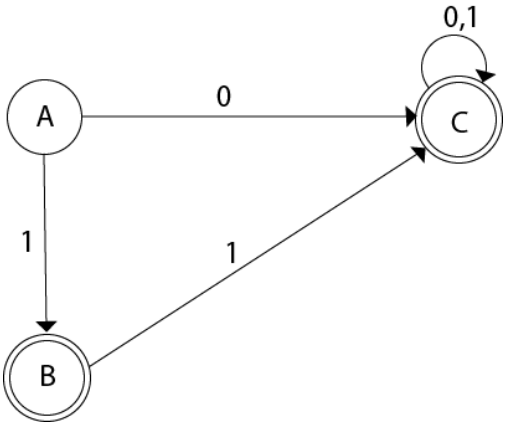


Figure 2.5: Illustration of equivalent NFA and DFA

2.4.3 Regular Expression to Finite Automaton Conversion

A traditional way to solve regular expressions is to convert them to finite state automatons. [1, 46, 24, 31] The conversion involves constructing a finite automaton that can recognize the same language as the given regular expression. The basic idea is to represent the regular expression's structure and operations in terms of states and transitions in the finite automaton. This process typically involves creating states for different components of the regular expression (e.g., individual characters, concatenation, alternation, and Kleene closure) and defining transitions between these states based on the relationships specified by the regular expression. For instance, concatenation is represented by connecting the accepted state of the first sub-expression to the start state of the second, denoted as S→S′. Alternation introduces branching paths, represented as S→S′ or S→S″, where different paths represent different possibilities. Similarly, the Kleene closure introduces looping paths, represented as S→S, allowing for repetition.

The resulting NFA can then be used to recognize whether a given input string belongs to the language described by the original regular expression. We can also convert the NFA to a DFA using the powerset construction [34] introduced by Rabin and Scott. While NFAs

are more expressive and easier to design, DFAs are more efficient for implementation and evaluation. We simplify the structure by converting an NFA to a DFA, eliminating non-deterministic transitions and redundant states. Figure 2.6 shows a pair of equivalent NFA and DFA of a given regular expression P.

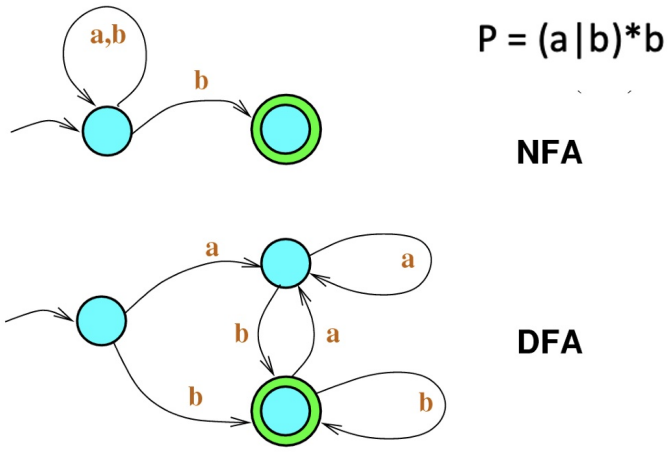


Figure 2.6: Illustration of RegEx conversion

In the context of General-Purpose Graphics Processing Unit (GPGPU) acceleration, the choice between DFA and NFA becomes critical. DFAs, with their deterministic and linear-time execution, are constrained by their serial nature, making them less suitable for leveraging GPU parallelism. On the other hand, NFAs, with their inherent non-determinism and potential for complex pattern recognition, offer a promising avenue for exploiting the parallel computing capabilities of GPUs. This research focuses on harnessing the parallelism of NFAs to address the limitations of DFAs and explore innovative approaches for pattern matching in GPGPU-accelerated environments.

Chapter 3

Related Work

We have found that some previous works on regular expression matching and literal string matching exist in different domains. In this chapter, we will provide an overview of them.

3.1 Regular Expression matching

We observe that not only GPUs but also the use of SIMD instructions with CPUs can achieve parallel computation for accelerated computation.

Sitaridi et al.(2016) [38] presented the design and implementation of SIMD vectorized regular expression matching for filtering string columns, which processes multiple input strings in a data-parallel way without accessing the input in lockstep. They used the vectorization technique to achieve data-parallel processing of multiple input instances, where accelerated database operations are executed on CPUs and Xeon Phi co-processors.

Hyperscan [12] is another CPU SIMD-based high-performance multiple regex matching library developed by Intel. Its fundamental principle is similar to this study, which can

be divided into compile-time and run-time. It comes with a regular expression compiler written in C++. As shown in Figure 3.1, Hyperscan takes regular expressions as input, transforms them into finite state automatons, and puts them into a database for later use by the runtime. The Hyperscan run-time is developed in C. Figure 3.2 shows a high-level block diagram of the main components of the run-time; "datablocks" is the input corpus, and users use the compiled database of REs to call Hyperscan's scan APIs to trigger internal matching engines (nondeterministic finite automaton (NFA), deterministic finite automaton (DFA), and so on) to match the corpus. When performing pattern matching, scratch space is used to store intermediate results and metadata necessary for processing the input data efficiently. This scratch space is dynamically allocated and managed by the Hyperscan library as needed during runtime. Hyperscan accelerates these engines with the help of single instruction, multiple data (SIMD) instructions provided by the Intel processor, and matches are delivered to the user application for processing via a user-provided callback function. [12]

In addition to SIMD, Hyperscan incorporates an optimization technique known as regular expression decomposition. This approach partitions a regular expression into a series of strings and finite automaton (FA) components. Consequently, regex matching is restructured into a series of subregex matches. In such a way, string matching is integrated into the regex matching process rather than serving merely as an initial trigger. This integrated approach, which diverges from prefilter-based designs, ensures continuous tracking of string-matching states, thereby precluding superfluous operations. Another benefit is that decomposition typically results in smaller FA components, which are more amenable to conversion into deterministic finite automaton (DFA) and thus benefit from fast DFA matching [41]. In the latter part of this thesis, we will compare Hyperscan with our algorithm and provide a detailed analysis.

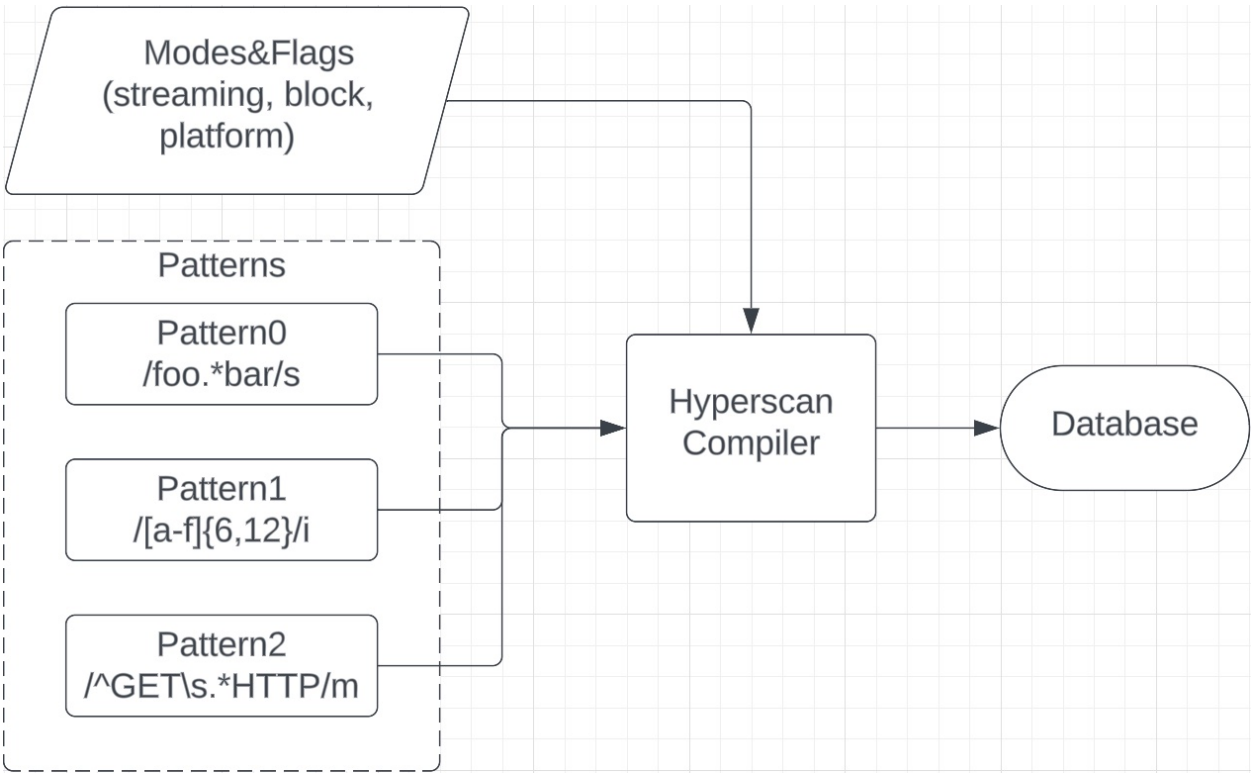


Figure 3.1: Hyperscan compilation process [11]

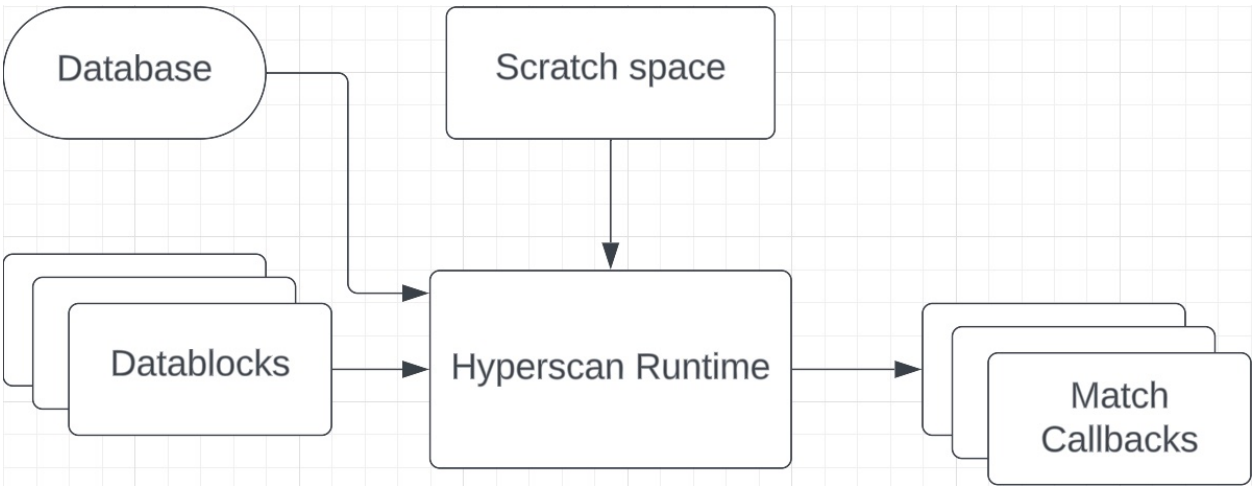


Figure 3.2: Hyperscan run-time [11]

Regarding GPU algorithms, a traditional way to solve NFAs is to employ state-level parallelism. iNFAnt [7] is a well-known algorithm that turns regex patterns into NFAs and then solves them with CUDA. It uses state-level parallelism, which means it splits the tasks among different threads when it moves from one state to another while reading an input character. This needs the problem size to be big enough to maximize the GPU's power. However, in our experiments, a row in the transition table normally is less than 10 elements long. This stops it from benefiting from larger block sizes, lowering occupancy. Yu and Becchi proposed an optimized iNFAnt [44], which clusters NFA states into groups by certain rules[45] to reduce the size of the problem. However, it still faces the issue of low occupancy.

To fix iNFAnt's occupancy problem, Liu et al. (2023) developed ASyncAP [23]. It distributes the text to different blocks, and each thread tries to solve the current NFA from different starting points. If it can not find a match at some step, the thread goes on to the next starting point. For example, in figure 3.3, a nine-thread block is deployed to locate the sequence "def" within a given string "abcdefghijklmnop." Each column represents the input string for each thread. Gray blocks indicate the termination of a search, and the fourth column reports a match. More specifically, thread T1 starts from the initial character 'a', T2 from the second character 'b', and this pattern continues sequentially. Upon a failed attempt by T1, it advances to the tenth character, factoring in the block size (1+9). The process terminates with thread T4 identifying a match, prompting the kernel to output the result.

This way, we can improve with a larger thread block size and use the computational resources more efficiently. However, this method also brings problems with computational overhead. Its theoretical time complexity is $O(mn^2)$, where m is the NFA size, and n is the string length. Even though, in most cases, it will stop at the first character, if our regex pattern starts with a wildcard or, worse, a wildcard with a quantifier like *, the time com-

plexity can get to its worst case, making the computational workload much bigger and hurting the performance. We also point out that such regex patterns are not rare in different databases.

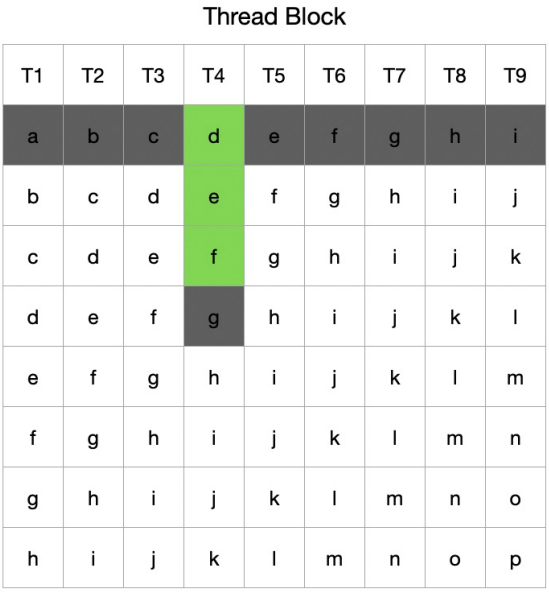


Figure 3.3: Illustration of the ASyncAP [19]

In addition to the software-based regular expression matching mentioned above, we have also recognized a category of high-performance regular matching methods based on hardware FPGA platforms [20, 43, 3]. The available hardware resources of the design imply the maximum size (in terms of the amount of states and transitions) of the supported automata. However, regular expressions may be arbitrarily complex. Therefore, we consider such methods supplementary, akin to the prefilter stage that will be mentioned later in this thesis, which serves as an accelerator rather than the matching engine.

3.2 Literal matching

Regex patterns offer powerful ways of expressing complex and flexible matching conditions but have much higher computational complexity than literal matching. Regex engines usually involve complex state transitions within finite automaton, which require a lot of resources. In contrast, literal matching involves simpler character-by-character comparison, a computationally less expensive task. To reduce the computational overhead of regex pattern matching, researchers and developers have investigated the idea of using literal matching as an initial "prefilter" stage. In this pre-filtering technique, Xu et al. (2023) [42] and Qiu et al. (2021) [33] have shown the impressive cost-effectiveness of literal matching compared to regex matching. The results in this thesis explain that literal matching, when carefully integrated into pattern-matching pipelines, achieves a performance advantage that is two orders of magnitude better than regex matching. This cost-effective and efficient strategy has been widely applied in Deep Packet Inspection (DPI) systems, as shown by popular applications such as Snort [35] and Suricata [29]. In this thesis, we will implement and compare two widely acknowledged literal pattern-matching techniques with linear time complexity using CUDA. Our primary focus is to perform a detailed performance comparison and analyze the underlying factors contributing to any observed differences. In this section, we will introduce the algorithms, reserving the comprehensive CUDA version implementation discussion for Chapter 4.

3.2.1 KMP

The Knuth-Morris-Pratt (KMP) algorithm [16] is a highly efficient string searching algorithm. The key to its efficiency lies in constructing a "failure function" built-in *ComputePrefix* that helps skip unnecessary character comparisons during the search process. Here is a brief introduction to the KMP algorithm in the following two pseudo-code blocks:

Algorithm 1: KMP algorithm

```
1  Algorithm KMP()
2      next ← ComputePrefix()
3      i ← 0
4      j ← 0
5      while i <len(string) do
6          if string[i] == pattern[j] then
7              i = i + 1
8              j = j + 1
9          else if j > 0 then
10             j = next[j − 1]
11         else
12             i = i + 1
13         if j == len(pattern) then
14             return true
15         end
16     end
17     return false
```

Purpose: This function is designed to find if the substring pattern exists in the given string.

Parameters:

pattern: The input substring.

string: The given string to be examined.

Returns: A boolean value indicates if the substring pattern exists in the given string.

When a mismatch occurs between the current character of the string and the current character of the pattern during the matching process, instead of restarting the comparison from the beginning of the pattern, the KMP algorithm uses the information stored in the "next" array returned from the ComputePrefix to determine how far to shift the pattern before continuing the comparison. This shift allows the algorithm to skip characters in the pattern that are guaranteed to match the already processed characters in the string.

The ComputePrefix method generates a "next" array that helps the Knuth-Morris-Pratt (KMP) algorithm efficiently handle mismatches. It calculates the longest proper prefix of the pattern, which is also a suffix. This information is then used during string matching to determine how many characters can be skipped in the pattern when a mismatch occurs, thus avoiding unnecessary comparisons and backtracking. Consider the pattern "ABABAC." The next array for this pattern is computed as follows: start with the first character 'A,' which has no proper prefix, so next$[0] = 0$. No proper prefix for 'B' matches a suffix, so next$[1] = 0$. For the third character 'A', the proper prefix "A" matches the suffix, so next$[2] = 1$. For the fourth character 'B', the proper prefix "AB" matches the suffix, so next$[3] = 2$. For the fifth character 'A', the proper prefix "ABA" matches the suffix, so next$[4] = 3$. For the last character 'C', there's no matching proper prefix, so next$[5] = 0$. Hence, the next array is $[0, 0, 1, 2, 3, 0]$.

By using the "next" array, the algorithm ensures that it never compares characters that have already been matched, thereby avoiding redundant comparisons and improving the overall efficiency of the string-matching process.

```
1  Procedure ComputePrefix()
2      next ← [0];
3      prefix_len ← 0;
4      i ← 1;
5      while i < len(pattern) do
6          if pattern[prefix_len]== pattern[i] then
7              prefix_len += 1;
8              next.append(prefix_len);
9              i += 1;
10         else
11             if prefix_len == 0 then
12                 next.append(0);
13                 i += 1;
14             else
15                 prefix_len = next[prefix_len - 1];
16             end
17         end
18     end
19     return next
```

Purpose: This function is designed to construct a prefix array from a given pattern for use by the KMP algorithm. It allows pattern matching to continue from the point of failure without restarting from the beginning of the pattern.

Parameters: pattern: The input pattern, same as KMP's input.

Returns: The constructed prefix array from the given pattern

Lin et al. (2013) proposed a parallel approach [21] based on patterns to implement CUDA KMP, where each thread takes a pattern from the pattern array and executes the KMP algorithm on the input string. This method works well when there are many patterns to look for. However, suppose the number of patterns is not enough. In that case, it becomes difficult to fully utilize the computational power of the GPU, especially in the context of the current extremely powerful GPUs.

3.2.2 Shift Or

The key idea behind the Shift-Or algorithm [9] is to maintain a 256-row mask table representing the pattern's characters since we are using ASCII encoding. However, this also means it will be limited by the encoding method. It converts literal matching into bitwise SHIFT and OR operations by left-shifting the state mask over the text, one bit per character, and updates the state mask according to the current character being examined. Typically, it works fine if the length of the search pattern should be short enough to fit into one machine word.

Algorithm 2: Shift Or algorithm

```
1  m = len(pattern)
2  pattern_mask = [∼ 0] × 256
3  for i ← 0 to m do
4  |   pattern_mask[pattern[i]] &= (1L << i)
5  end
6  R ←∼ 1
7  for i ← 0 to len(string) do
8  |   R = R| pattern_mask[string[i]]
9  |   R <<= 1
10 |   if (R&(1L << m)) == 0 then
11 |   |   return true
12 |   end
13 end
14 return false
```

Purpose: This function is designed to find if the substring pattern exists in the given string.

Parameters:

pattern: The input substring.

string: The given string to be examined.

Returns: A boolean value indicates if the substring pattern exists in the given string.

It initializes a mask table where each bit corresponds to a character in the pattern and clears the bits for characters in the pattern. Then, it iterates through the input string, updating a bit vector R to represent potential matches. If a potential match is found, it returns true;

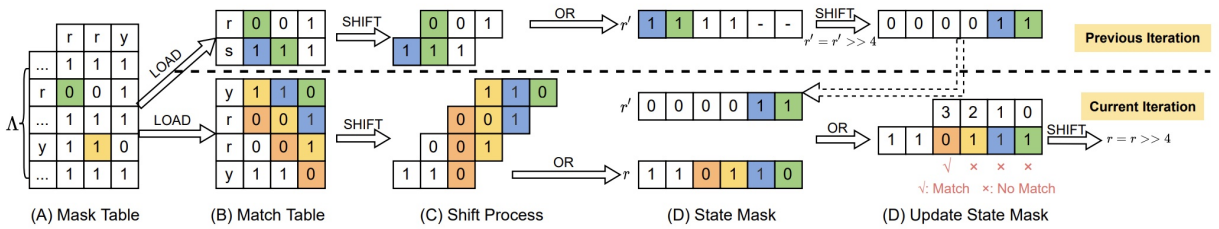


Figure 3.4: Shift or algorithm [42]

otherwise, it returns false. This approach avoids redundant comparisons by leveraging bitwise operations to track matches efficiently. Consider the example in figure 3.4, where the input string is 'rsyrry', with 'rs' being processed in the previous iteration and 'yrry' in the current iteration, and 'rry' be our pattern:

The mask table has 256 rows and pattern-length columns. Each row vector represents an ASCII character. For each column in each row, we set it to 0 if the ASCII character equals the corresponding character in the pattern. For example, row 'r' is 001 since the first two characters of the pattern 'rry' are 'r.' After constructing the mask table, the system starts processing the input string in chunks called iterations. During each iteration, it handles a specific number of input characters. For each character c in an iteration, the corresponding entry from the mask table is loaded into the match table. For instance, if the system processes 4 characters per iteration, the match table is illustrated in Fig 3.4(B). To obtain the match results, the Shift-Or algorithm shifts the i^{th} character in the input string left by i bits to align the bits diagonally. As illustrated in Figure 3.4(C), during the current iteration, the four vectors are shifted left by 0, 1, 2, and 3 bits, respectively. The vectors were similarly shifted in the previous iteration. Then, it performs an OR operation on the shifted vectors to obtain the state mask and check if there is a match, as illustrated in Figure 3.4(D).

Chapter 4

Algorithm Details

In this chapter, we will introduce the limitations of regular expressions and explore how to optimize these limitations by utilizing pre-filtering techniques. Subsequently, we will dive into the specific implementation algorithms for the two matching types. More specifically, we will present CUDA-KMP, CUDA-ShiftOr, CUDA naive matching algorithms for literal matching, and optimized ASyncAP algorithm for regular expression matching.

4.1 Literal matching for prefiltering stage

For this stage, there are existing approaches based on SIMD-CPU algorithms based on the FPGA (field-programmable gate array) platform [37, 36, 15]. Given the relatively low algorithmic complexity of string matching and GPUs having more computing resources, we will implement CUDA-based algorithms. We will extract the longest static sub-string from each RE pattern for the pre-filtering patterns. For example, *"configName="* would be the pre-filtering pattern for RE pattern `[?&]configName=[^&]+(script|onload|src)`. If a RE only contains special symbols, we will skip the pre-filtering stage.

Note that all three following algorithms share the same purpose, parameters, and returns:

Purpose: This CUDA function is designed to find if the substring pattern exists in the given string array parallelly.

Parameters:

strings: The given string array to be examined.

strings_length: The corresponding length of each string in the string array.

threadIdx: The current thread index, which is a 3-D vector. We consider threadIdx.x the index since we use 1-D thread blocks.

blockDim: The dimensions of the thread block, which is a 3-D vector. We consider blockDim.x the size since we use 1-D thread blocks.

Built-in keywords and functions:

__shared__: indicates the parameter is shared among the same thread block.

__syncthreads(): threads will be halted until all threads in the same thread block reach this line.

Returns: A boolean value indicates if the substring pattern exists in the given string.

4.1.1 CUDA-KMP

To address the limitation mentioned in section 3.2.1, we implemented an optimized version of CUDA-KMP [21], which is a string-based parallel approach in which each thread is assigned a segment of the input string for pattern matching. For short inputs, we can make the strings long enough to use the GPU's computational power using string buffering techniques. Each sub-string begins from $k \times n$ where $k \in \mathbb{N}$ and $n =$ pattern length,

and is $2\times$ pattern length long. This way, we can be sure to catch the parts that straddle two neighboring sub-strings.

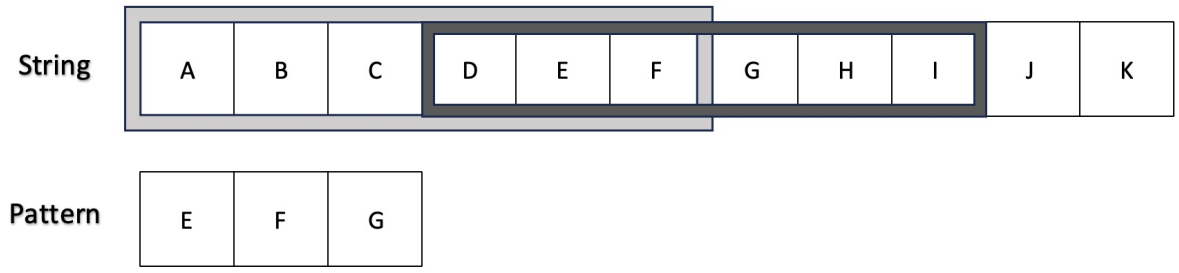


Figure 4.1: Illustration of the string-based task distribution [19]

Figure 4.1 illustrates an example, looking for the pattern "EFG" in the input string. Each thread will scan the string from a different location, and a thread might fail to spot the pattern if it is split across sub-string boundaries. For example, if each thread takes 3 characters, "EFG" would be split by the first and second threads. Each thread's search has extra padding based on the pattern length to avoid this. A thread gets a sub-string that overlaps with its neighbor(s) by the same amount as the pattern length, which is 6 in this case. This way, we can be sure that a sliding window of the same size as the pattern length, moving from left to right, will always fall within at least one thread's sub-string. Since our literal patterns are much tinier than the input strings, the slight redundancy is worth it because we do not need to worry about finding incomplete pattern instances on sub-string boundaries. The pseudocode below shows how the algorithm works in detail.

```
1  Algorithm CUDAKMP()
2      target ← strings[blockIdx.x]
3      m ← strings_length[blockIdx.x]
4      stride ← blockDim.x
5      next ← ComputePrefix()
6      for index ← threadIdx.x to m by stride do
7          i ← n× index ; j ← n× (index +2) - 1
8          if i > m then
9              return
10         end
11         if j > m then
12             j = m
13         end
14         k ← 0
15         while i < j do
16             if target[i] == pattern[k] then
17                 i + +; k + +
18                 if k == n then
19                     return true
20                 end
21             else if k > 0 then
22                 k = next[k − 1]
23             else
24                 i+ = 1
25         end
26     end
27     return false
```

4.1.2 CUDA-Shift Or

First, we implemented the CUDA version of the traditional shift-or algorithm. We used a state mask of the length of a machine word (64-bit on the machine used in this thesis). Because we need to shift the state mask to the left by pattern length bits, we can only check 64-pattern length characters simultaneously in each iteration.

Due to the limitation of the traditional Shift-Or algorithm, the 64-pattern length restriction in each iteration greatly reduced the GPU's occupancy, which means limiting the use of GPU hardware performance, especially when the pattern length is too long. To solve this problem, we proposed CUDA-ShiftOr-Optimized. We use five 64-bit numbers to represent the current state mask, the smallest number sufficient for all the cases in our database. It greatly increases the amount of parallel computation in each iteration. It also greatly increases the upper limit of the pattern length. The following pseudocode only provides the traditional version of the CUDA-ShiftOr algorithm, and the Optimized version is only to change its state mask to a 64-bit word array. Note that __syncthreads() is used for syncing the shared memory variables to prevent race conditions.

Algorithm 3: Shift-Or Matching

```
1  target ← strings[blockIdx.x]
2  m ← strings_length[blockIdx.x]
3  stride ← blockDim.x
4  __shared__ unsigned long long prev
5  __shared__ unsigned long long curr_mask
6  if threadIdx.x == 0 then
7  |   prev ← 0
8  end
9  __syncthreads()
10 curr_pos ← threadIdx.x
11 while curr_pos < str_len do
12 |   curr_mask ← prev >> (65 − pat_len)
13 |   curr_mask| = mask_table[string[curr_pos]] << threadIdx.x
14 |   __syncthreads()
15 |   prev ← 0
16 |   __syncthreads()
17 |   atomicOr(&prev, curr_mask)
18 |   __syncthreads()
19 |   if (prev&(1 << threadIdx.x)) then
20 |   |   return true
21 |   end
22 |   curr_pos+ = stride
23 end
```

4.1.3 CUDA naive matching

As a control group, we set up a naive matching algorithm. The task distribution is similar to CUDA-KMP, and in each thread, we start from the corresponding starting position and try to match the pattern. If it fails, we increment the starting position by blockDim.x characters. The detailed logic is shown in the pseudocode below:

Algorithm 4: CUDA Naive Matching

```
1  target ← strings[blockIdx.x]
2  m ← strings_length[blockIdx.x]
3  stride ← blockDim.x
4  for index ← threadIdx.x to m - pat_length by stride do
5      counter ← 0
6      while index < pat_length and target[index + counter] == pattern[counter] do
7          counter += 1
8      end
9      if counter == pat_length then
10         return true
11     end
12 end
13 return false
```

4.2 Regular expression matching

We introduce an optimization technique for the performance issue that occurs when RE starts with wildcard elements. This technique consists of annotating regex patterns that greatly affect performance. For example, when the NFA derived from the regex pattern has many transitions from the initial state to other states, we eliminate the first or some

of the easily reachable states and set a more challenging state as the initial one. Then we validate if the part we eliminated can be matched after obtaining a successful match.

Consider the worst case when the RE starts with a ".*". Unlike instances where threads terminate after inspecting the initial characters, this wildcard scenario compels each thread to persist in a matching loop until the text corpus is fully scanned, potentially escalating the computational load significantly. Our preliminary trials indicated that this issue negates the performance benefits of the ASyncAP method, making it worse than the performance of CPU single-thread matching.

While compiling an RE into a Non-Deterministic Finite Automaton (NFA), we record the in-degree for each state. States with an in-degree surpassing a predetermined threshold (100 in our study) are classified as easy-to-reach states. We introduce the concept of a "skip state" for any easy-to-reach states that is in one of the following two situations:

1. Connected to the initial state 0

2. Connected to another skip state

Here is a simple example of two skip states (marked as green) in Figure 4.2:

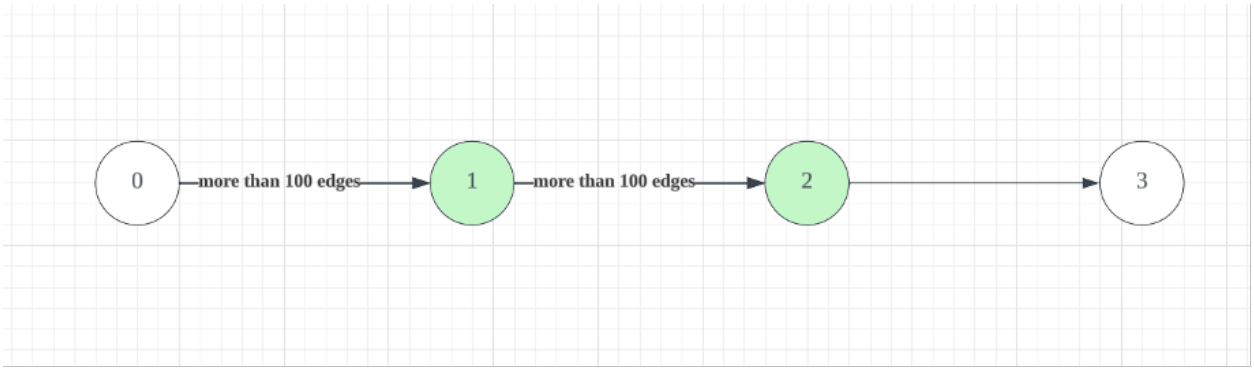


Figure 4.2: Skip state example. State 0 is the initial state, and states 1 and 2 (in green) are identified as skip states

Subsequently, we derive a reduced NFA from the original, ensuring the initial state is disconnected from any easy-to-reach states. The sub-NFA, extracted from the original NFA, is responsible for the final screening after the reduced NFA finds a match.

For instance, take the regex pattern: `[^abc]+https?://t.com`. We first examine if it can match `https?://t.com`. If it does, we will continue to look for other matches if the character before it is one of `abc` or report a match found if it is not.

Regarding the data structure representation of NFA, ASyncAP [23] opted for a simple transition table. This table might be sparse and inefficient in memory usage, and Blaß and Philippsen [4] suggest sparse representations, such as *COO*, *ELL* or others as better alternatives to represent such graphs. Other works [22, 25] proposed another per-node structure similar to Compressed Sparse Rows (CSR). However, it cannot handle large-scale problems with many states. In the specific case of regular expression matching, the situation is distinct. Assuming a simple ASCII representation, we have a constant number of nodes, the (at most) 256 characters in the ASCII table. Hence, we still select an alphabetical representation similar to the transition table, but we only store the edges, which represent transitions among different states that exist in the array. This is clearly not suitable when patterns contain characters from various alphabets, but it has the benefit of ensuring efficient access in the many cases when ASCII is adequate.

Chapter 5

Dataset, methodologies, and experiments

In this chapter, we will introduce the methods employed for data acquisition, the characteristics of the data, the experimental approach, and the results obtained. Furthermore, we will delve into the implications conveyed by the experimental results.

The following describes the primary experimental machine we used to test the performance differences between various algorithms.

CPU	Intel Core i5-10500H
Memory	DDR4 8GB x 2
GPU	NVidia GeForce RTX 3060 Laptop

Table 5.1: Primary hardware specifications

Additionally, to compare the potential impact of different hardware on performance, we

also utilized the machine listed below for comparison. Note that RTX 3080s share the same Ampere architecture and memory size as our primary experimental machine but with more computing power (SM count, tensor core count, etc).

CPU	Intel Core i5-9400F
Memory	DDR4 8GB x 2
GPU	NVidia GeForce RTX 3080

Table 5.2: Comparison hardware specifications

OS	Ubuntu 22.04.3 LTS
Driver	NVIDIA Display Driver version 535.104.05
SDK	CUDA 12.2
Tool	NSight Compute 2023.2.2 [26]

Table 5.3: Software specifications

5.1 Dataset and its summary statistics

The hardware and software specs are in Tables 5.1, 5.2, and 5.3. We could not use the test suite that iNFAnt picked for the dataset because it is no longer available. The corpus [39] that ASyncAP used are also binary files incompatible with our tests. To compare our work with other approaches of RE matching on GPUs for deep packet inspection, we used regex patterns from Snort [35, 40], as ASyncAP did, and made a synthetic corpus with mostly HTTP code, SQL queries, as input using the ChatGPT API [17]. In addition, we have generated random strings of the same length to serve as a control group to investigate the

potential impact of different types of corpora on the experimental results. We tested how each algorithm performed using NSight Compute 2023.2.2 [26].

5.1.1 Regex Patterns

To explore the impact of different regex patterns on experimental results, we categorized the regex patterns based on the number of transitions in their compiled representations into S (fewer than 100 transitions), M (100 to 500 transitions), L (500 to 1000 transitions), and XL (more than 1000 transitions). Noting that whether a pattern begins with a wildcard symbol (or a regex element similar to a wildcard, such as `[^a]`, significantly affects the performance of ASyncAP, we also classified regex patterns based on whether they have more than 100 transitions from the initial state to the next state to determine if they begin with a wildcard.

Furthermore, to investigate the potential effects of different regular expressions, we also employed regular expressions from ClamAV [8] and Dotstar [2]. Although all three regular expression datasets are designed for deep packet inspection, the regular expressions within them exhibit different characteristics. For instance, we consider the two most important features to be the average size of each regular expression in the datasets and the average in-degree per node. We will investigate the impact of these features on performance.

5.1.2 GPT Corpus Generator

We observed a lack of practical, universally applicable corpora for string matching testing at present. The manual collection of extensive corpus materials is both time-consuming and impractical. Therefore, we developed a simple tool called the GPT Corpus Generator, which is a Python script utilizing the API provided by OpenAI for ChatGPT[30]. This tool

enables us to issue instructions for the desired content of the corpus. In practice, we set the background as 'You are a helpful assistant who understands data science and computer science' and the instruction as 'Give me 20 SQL query command code blocks for string matching testing purposes.' 'Give me 20 HTML code blocks for string matching testing purposes.' and so on. Then, save the contents of the code blocks from the response locally. This way, we can obtain corpus content of any desired type. It is important to note that due to the length limitation of GPT-generated responses, excessive text content should not be requested in instructions. This limitation can be addressed by running more iterations. The generated corpus is around 20 Megabytes and can be found in the repository of this work [18].

5.2 Tuning Compilation Configurations and Running Settings

This section will present the optimal configurations for the algorithm's performance, which depends on some compilation and runtime parameters. We will also explain how we used NSight Compute [26] to profile the algorithm and find the optimal configurations. The following subsections will discuss the details of each parameter.

5.2.1 Register Counts per Thread

One important parameter is the number of registers per thread. More registers per thread means less memory access, as more data can be stored in registers. However, more registers per thread means fewer threads can run simultaneously on each streaming multiprocessor (SM) because each SM has only 65,536 32-bit registers.

For example, on our experimental machine (see table 5.1), the default number of registers

per thread is 74. If we use 1024 threads per block, then 1024 * 74 > 65,536, so we do not have enough registers to run the kernel, and the program will crash.

So, we must find the best number of registers per thread that maximizes occupancy. Occupancy is the ratio of active threads to the maximum possible threads on each SM. Although we can calculate the occupancy, we can also use the profiler NSightCompute to see how the register count affects occupancy conveniently. Figure 5.1 (top) shows the "Impact of Varying Registers Per Thread" section, where we can see the highest number of registers still giving us the highest occupancy.

5.2.2 Block Size

Another parameter that affects occupancy and performance is the block size. If the block size is too small, we may waste some threads on the SM because the shared memory size limits the number of blocks that can run simultaneously. If the block size is too large, we may have some idle threads because the blocks may not fit well on the SM. But if the block size is right, we may get a theoretical occupancy of 100%.

To understand this, imagine filling a big 2m x 2m square with smaller squares. If we use 0.5m squares, we can fill it exactly. If we use 0.6m squares, we will have some gaps. If we use 1m squares, we can fill it exactly again. Figure 5.1 (middle) shows our algorithm's best block size in the "Impact of Varying Block Size" section.

5.2.3 Shared Memory Usage per Block

The last parameter we will discuss is the shared memory usage per block. As we said before, each SM has a limited amount of shared memory. So, the more shared memory each block uses, the fewer blocks can run simultaneously. When we design algorithms, we need to balance the block size and the shared memory usage per block. Figure 5.1 (bot-

tom) shows the optimal configuration in the "Impact of Varying Shared Memory Usage per Block" section.

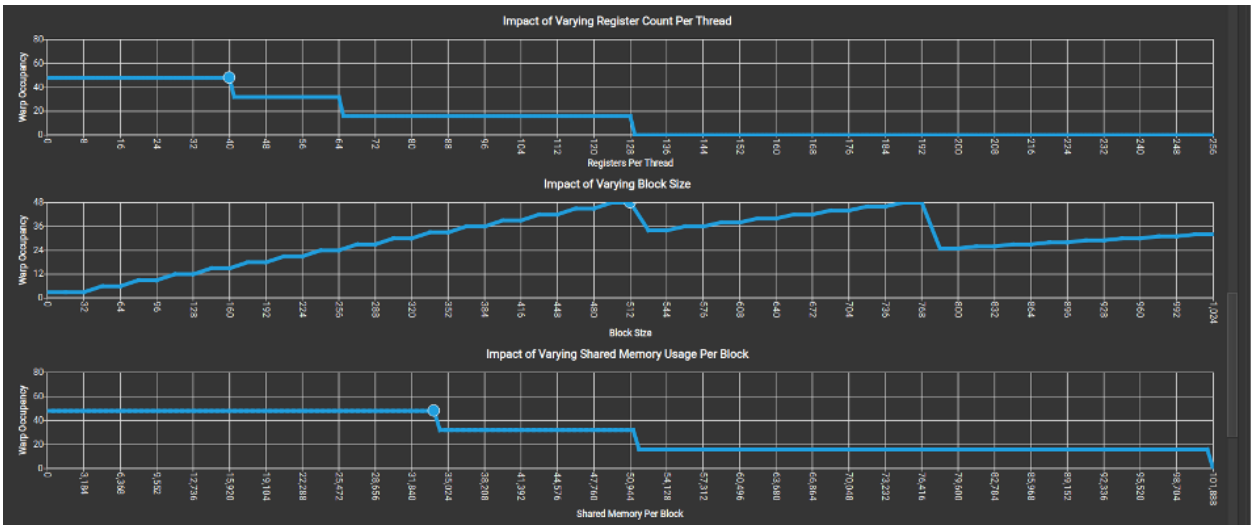


Figure 5.1: NSight Compute [26] Occupancy Analysis

All three y-axes in figure 5.1 represent the warp occupancy. The x-axes of the three diagrams represent register count per thread, block size, and shared memory, respectively. It shows how the occupancy changes while the other three factors change.

5.3 Methodologies and Results

This section will introduce each part of the experiment in detail and explain how we designed the experiment to achieve its purpose.

5.3.1 Literal pattern matching

We obtain the longest string without special symbols from the regex pattern as the prefilter pattern in the regex compilation process, as stated in section 4.1. This is independent of the length of the regex pattern. Also, the length of the string, not the length of the pattern,

determines the time complexity of KMP. Hence, we do not perform experiments with the length and type of regex as variables.

The pattern-based and string-based methods differ mainly in that the string-based method can employ a buffer to secure a large enough block size, whereas the pattern-based method cannot ensure it when the number of patterns is small. Thus, we only examine the effect of different block sizes on occupancy under the same text.

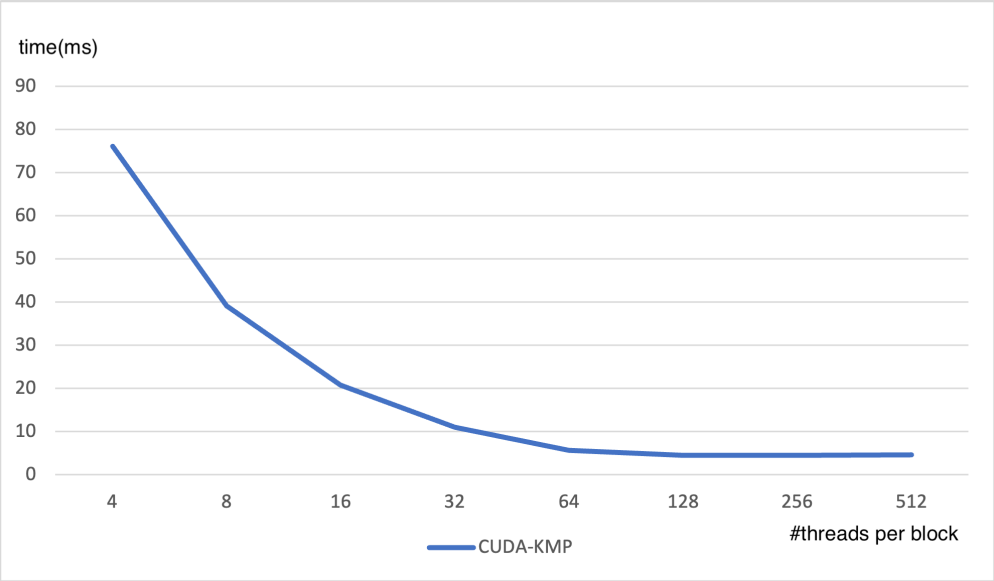


Figure 5.2: CUDA-KMP performance over #threads/block

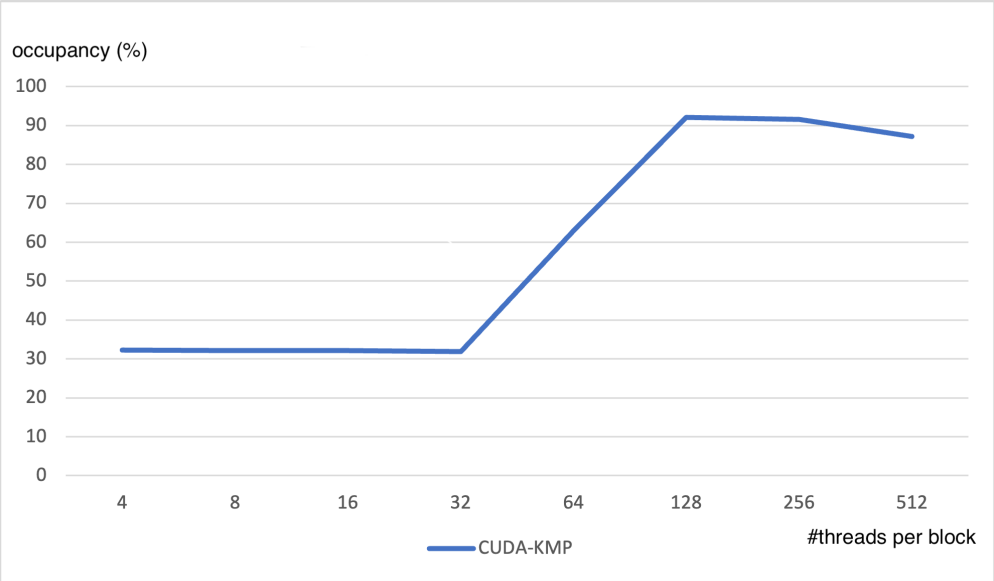


Figure 5.3: CUDA-KMP occupancy over #threads/block

The experimental results shown in figures 5.2 As figures 5.2 and 5.3 demonstrate, the block size of the pattern-based method is insufficient when the number of regex patterns is less than or equal to 64, and only a limited number of blocks can be executed on the same SM, leading to a low occupancy. The pattern-based method can only reach the maximum occupancy when the number of regex patterns is at least 128. Hence, our method has a remarkable benefit of about 2x-40x acceleration compared to the pattern-based method when the number of regex patterns is below 128 (varying with the different number of regex patterns).

CPU-KMP	1823.25ms
CUDA-KMP	3.45ms
CUDA-Naive	2.05ms
CUDA-ShiftOr	41.03ms
CUDA-ShiftOrOptimized	21.02ms

Table 5.4: CPU vs CUDA

We also conducted a CPU version experiment to emphasize the benefits of CUDA computation. Since we could not independently assess the CPU version of the KMP algorithm using NSight Compute [26], we depended on the runtime measurements from the code for both the CPU and CUDA versions. For the CUDA version, this also involves the extra overhead of calling GPU methods. As shown in table 5.4, all CUDA algorithms still achieve about a 44x to 900x speedup compared to the CPU version.

5.3.2 Regular expression matching

For the regular expression matching part, to show how we can benefit from the parallelism of GPU-based algorithms, we used a CPU single-thread version as a control group to evaluate the performance of iNFAnt, ASyncAP, and ASyncAP-Optimized, where ASyncAP-Optimized is our proposed method. For this experiment, we classified the regex patterns into three groups: small (less than 100 edges), medium (between 100 and 500 edges), and large (more than 500 edges), where edges denote transitions between different states. The experimental results are displayed in the tables 5.5, 5.6, and 5.7. Moreover, we split them into two categories based on whether the initial state is 0 (indicating it begins with a wildcard) or not. These results are presented for the larger RE patterns in tables 5.8 and 5.9. The time consumption in the tables refers to the average time for each regex

pattern to match the corpus among 100 regex patterns to get a more accurate and stable performance metric that is less likely to be skewed by anomalies or outliers. Note that we did not apply NSight Compute [26] to measure their performance for Table 5.9, because NSight Compute [26] demands multiple iterations to average test performance, and ASyncAP performed so badly in this set of tests that the use of NSight Compute [26] became impractical.

CPU	5028ms
iNFAnt	768ms
ASyncAP	18ms
ASyncAP-Optimized	20ms

Table 5.5: Performance with small regex patterns

CPU	5523ms
iNFAnt	810ms
ASyncAP	25ms
ASyncAP-Optimized	26ms

Table 5.6: Performance with medium regex patterns

CPU	7223ms
iNFAnt	1660ms
ASyncAP	39ms
ASyncAP-Optimized	42ms

Table 5.7: Performance with large regex patterns

CPU	7523ms
iNFAnt	1301ms
ASyncAP	38ms
ASyncAP-Optimized	40ms

Table 5.8: Performance not starting with wildcard regex patterns (large size)

CPU	7334ms
iNFAnt	1218ms
ASyncAP	71946ms
ASyncAP-Optimized	38ms

Table 5.9: Performance starting with wildcard regex patterns (large size)

RE_ID	ASyncAP	ASyncAP_Optimized
184	59886ms	24ms
1637	129885ms	36ms
2421	33456ms	65ms
19	123289ms	36ms
2416	33453ms	65ms
1584	153479ms	59ms
266	128636ms	26ms
1815	525ms	27ms
197	536ms	27ms
1802	56307ms	15ms

Table 5.10: Detailed ASyncAP vs ASyncAP_Optimized

iNFAnt	983ms
ASyncAP	12ms
ASyncAP-Optimized	12ms

Table 5.11: Performance with large regex patterns, 3080

In our research, we added different groups to test how well the pre-filtering technology works and compare the overall performance with Hyperscan [12]. We tested the performance of Hyperscan [12] according to the Hyperscan Guide [13] and used hsbench [14] as the benchmark.

	S	M	L
iNFAnt	84.6ms	113.5ms	149.8ms
ASyncAP	10.1ms	14.2ms	16.8ms
ASyncAP_Optimized	10.2ms	14.2ms	17ms
Hyperscan	3.2ms	3.8ms	5.3ms

Table 5.12: RE matching with pre-filtering

For the experiments with different regular expressions, we randomly selected a total of 300 regular expressions from three groups—Snort, ClamAV, and Dotstar—evenly distributed in three different size categories: S, M, and L. We then calculated the average size and average in-degree per state for regular expressions of different categories. Table 5.13 shows the statistics of the regular expression groups and the experimental result:

	average size	average in-degree	performance
ClamAV	107	7.12	24ms
Snort	296	20.60	32ms
Dotstar	242	7.71	29ms
Teakettle	57	12.64	18ms

Table 5.13: Stats of different RE groups and performance

5.4 Summary and Discussion

This chapter will summarize the above experimental results, identify numerical patterns, and discuss the underlying causes.

In the Literal pattern matching part, we can see that although all the CUDA algorithms far outperform the CPU version, naive matching is the best performing algorithm, with about 40% performance improvement compared to the CUDA-KMP algorithm. It has about 10x to 20x speedup compared to the two shift-or algorithms. As mentioned in 4.1.2, the traditional shift-or algorithm performs poorly due to its low occupancy, which has a theoretical maximum of 33% and only reaches about 30% in practice, meaning that we only used 30% of the computing power. Shift-or optimized greatly increases the number of simultaneous computations in each iteration, reaching a theoretical occupancy value of 100% and achieving about 90% in practice, but because we need to check the state of the state mask in memory to determine whether the result has been found, more memory read and write becomes its performance bottleneck. The KMP and the naive algorithms, while achieving 90% occupancy, only need to check whether the current matching position has reached the pattern length without performing memory read and write. Let m be the length of the pattern, and n be the length of the string, here we find that the theoretically higher time complexity $O(mn)$ naive algorithm performs better than the lower time complexity $O(n)$ KMP algorithm, because when the hit rate is low, the 6th to 8th lines of the naive algorithm 4 are rarely executed, making the algorithm more converged to $O(n)$, and due to the simplicity of the algorithm, it only needs to execute fewer instructions, thus obtaining better performance. If the 6th to 8th lines of the naive algorithm 4 are executed a lot, consider the following situation, in which a pattern almost matches a string:

pattern: `abcdeabcdeabcde`

string: `abcdabcdacbdabcd`

In this worst-case scenario, every time `a` is checked, the naive algorithm will try to match to `d` and then fail. Thus, its time complexity is $O(mn)$. We designed a control group with a modified corpus according to the patterns to reach the worst-case scenario. We

obtained the results shown in table 5.14 to verify our conclusion. Even though the worst-case scenario would slow the performance, we observed that this situation rarely occurred and never happened in our experiments. Therefore, we consider the CUDA-Naive as a better method in the pre-filtering stage, at least for our dataset.

CUDA-KMP	3.45ms
CUDA-Naive	2.05ms
CUDA-Naive-WorstCase	5.03ms

Table 5.14: CPU vs CUDA

In the regular expression matching part, in most cases, ASyncAP and ASyncAP-Optimized both achieve about 200x and 30x to 40x performance optimization for the CPU version and iNFAnt respectively. However, when the regex starts with a wildcard, ASyncAP will have serious performance issues, more than 10 times worse than the CPU version. Table 5.10 shows a detailed ASyncAP vs ASyncAP_Optimized with RE IDs. At the same time, ASyncAP-Optimized solves this problem, and its performance is unaffected. We found that the traditional method of using state-level parallelism, such as the iNFAnt algorithm, has a smaller theoretical time complexity of $O(n)$ (because for each character, at most 256 transitions need to be traversed). However, the recently proposed ASyncAP has a worst-case time complexity of $O(n^2)$. Still, it has better utilization of GPU computing power based on different positions as the beginning of the string; when the regular expression does not start with a wildcard, the time complexity of ASyncAP converges to $O(n)$. Therefore, in most cases, ASyncAP performs better than iNFAnt. For its worst case, our optimization can eliminate most situations that make the time complexity of ASyncAP converge to $O(n^2)$.

As shown in table 5.12, we found that applying pre-filtering removes most of the workloads needed for matching regular expressions, making the time taken by different algorithms and RE sizes more similar. This means algorithms that usually do not perform well, like iNFAnt, get a bigger speed boost—about 10 times faster. The speed increase for ASyncAP and its optimized version is smaller but still about 8 times faster than iNFAnt.

In table 5.12, we also noticed that Hyperscan [12] performed very well and outperformed all GPU algorithms, including our optimized ASyncAP with pre-filtering. According to NSight Compute, the reason why we underperformed Hyperscan and the bottleneck of our algorithm is the bank conflict, which caused an 80% performance decrease. In CUDA, a bank conflict occurs when multiple threads in a warp access data from the same memory bank in shared memory simultaneously. Since each memory bank can only service one request per clock cycle, if multiple threads need data from the same bank, they must wait in line, which causes a delay. For fast visiting, however, we stored the state vector in shared memory, and all the threads were reading/writing from/to it simultaneously. Since the space of shared memory is limited, we do not have extra space to use to avoid the bank conflict. In addition to bank conflicts, since NSightCompute only accounts for the algorithms' own execution time, another performance discrepancy not reflected here is that our GPU algorithm requires approximately 8ms of overhead per regular expression to launch the kernel, memory transfer, etc.

We have also observed that in the control group of random string corpora, referred to in Section 5.1, there is no significant performance variance among the algorithms compared to the experimental group. Consequently, we have omitted the repetition of their experimental results data here. From this, we infer that the corpus length is the sole variable influencing the algorithms' time consumption.

For different hardware, we observed a 2-3 times faster performance for RTX 3080 from

table 5.11.

As for different regular expression groups shown in table 5.13, we found that the performance shows a significant difference when the average in-degree is similar, but the average size is significantly different, as with the ClamAV and Dotstar groups. Conversely, the performance is much closer when the average size is similar, but the average in-degree is significantly different, as with the ClamAV and Snort groups. Based on the data in table 5.13, we believe that the average size has a greater impact on performance within different groups of regular expressions. In contrast, the average in-degree is a factor that can be disregarded, especially after our optimized version of ASyncAP has addressed the issue with the leading wildcard.

Chapter 6

Conclusion and Future work

In the thesis presented, we comprehensively analyzed multiple methods employed for literal and regular expression pattern matching. The optimization techniques applied resulted in a marked enhancement of these methods, notably in their resilience and flexibility across diverse search contexts. Our empirical assessment substantiates the superior efficacy of CUDA programming over traditional GPU-based and sequential approaches in addressing pattern-matching challenges. Concurrently, our findings indicate that specific compilation and execution parameters considerably influence algorithmic efficiency. This observation underscores the need for meticulous parameter tuning to leverage performance gains fully.

To advance the field of regular expression (RE) matching, future research could benefit from deploying our methodologies on cutting-edge hardware to assess their robustness and transferability. Although NFAs can offer more parallelism, they also cause bank conflict, as we need a state vector to keep track of the states. Breaking the finite automaton and converting the NFAs to DFAs like Hyperscan could avoid the problem. Applying these optimizations and exploring DFA-based methods may be worthwhile. Delving into the de-

velopment of self-adjusting mechanisms for fine-tuning parameters may reveal methods to augment the flexibility of our system. The success of literal pre-filtering suggests that we can include simpler sub-patterns in our searches, making the matching process more efficient. Enriching our experimental scope to encompass REs from disparate sectors, including those employed in web engine Document Object Model (DOM) searches, might show the full extent of our methodology's adaptability. Moreover, given that this thesis has only identified regular expression datasets in the field of Internet packet inspection, it is valuable to explore datasets from other domains. Changing the algorithm from executing each regular expression individually to transferring all regular expressions to the GPU at once may significantly reduce the overhead caused by kernel launches and memory transfers. Finally, embedding our system within actual applications and evaluating its real-world performance could provide invaluable practical insights.

Bibliography

[1] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. 2016. Scalable Algorithms for NFA Multi-Striding and NFA-Based Deep Packet Inspection on GPUs. *IEEE/ACM Transactions on Networking* 24, 3 (2016), 1704–1717. https://doi.org/10.1109/TNET.2015.2429918

[2] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *2008 IEEE International Symposium on Workload Characterization*. 79–89. https://doi.org/10.1109/IISWC.2008.4636093

[3] Andreas Becher, Stefan Wildermann, and Jürgen Teich. 2018. Optimistic regular expression matching on FPGAs for near-data processing. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) (*DAMON '18*). Association for Computing Machinery, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/3211922.3211926

[4] Thorsten Blaß and Michael Philippsen. 2019. Which Graph Representation to Select for Static Graph-Algorithms on a CUDA-Capable GPU. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs* (Providence, RI, USA) (*GPGPU '19*). Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/3300053.3319416

[5] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. 2014. Deep Packet

Inspection as a Service. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) (*CoNEXT '14*). Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/2674005.2674984

[6] Robert D. Cameron. 1999. Perl Style Regular Expressions in Prolog, retrieved April 30, 2024. https://www.cs.sfu.ca/~cameron/Teaching/384/99-3/regexp-plg.html.

[7] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Comput. Commun. Rev.* 40, 5 (oct 2010), 20–26. https://doi.org/10.1145/1880153.1880157

[8] Cisco. 2024. clamAV, retrieved April 30, 2024. https://www.clamav.net/.

[9] Bálint Dömölki. 1968. A universal compiler system based on production rules. *BIT Numerical Mathematics* 8, 4 (01 Dec 1968), 262–275. https://doi.org/10.1007/BF01933436

[10] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.* C-21, 9 (1972), 948–960. https://doi.org/10.1109/TC.1972.5009071

[11] Intel. 2017. HyperScan Introduction, retrieved April 30, 2024. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html.

[12] Intel. 2017. HyperScan, retrieved April 30, 2024. https://github.com/intel/hyperscan.

[13] Intel. 2023. HyperScan Guide, retrieved April 30, 2024. https://intel.github.io/hyperscan/dev-reference/.

[14] Intel. 2023. HyperScan hsbench, retrieved April 30, 2024. https://www.intel.com/content/www/us/en/collections/libraries/hyperscan/performance-analysis-hyperscan-hsbench.html.

[15] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Detailed Characterization of Deep Neural Networks on GPUs and FPGAs. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs* (Providence, RI, USA) (*GPGPU '19*). Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/3300053.3319418

[16] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. https://doi.org/10.1137/0206024 arXiv:https://doi.org/10.1137/0206024

[17] Cheng Li. 2023. Test suite, retrieved April 30, 2024. https://github.com/cli117/thesis_work/blob/main/iNFAnt_Buffer/test_suite/midstr_7k.txt.

[18] Cheng Li. 2023. Testing corpus, retrieved April 30, 2024. https://github.com/cli117/thesis_work/blob/main/iNFAnt/test_suite/midstr_7k.txt.

[19] Cheng Li and Clark Verbrugge. 2024. Regular Expressions on Modern GPGPUs. In *Proceedings of the 16th Workshop on General Purpose Processing Using GPU* (Edinburgh, United Kingdom) (*GPGPU '24*). Association for Computing Machinery, New York, NY, USA, 26–32. https://doi.org/10.1145/3649411.3649416

[20] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. 2007. Optimization of Pattern Matching Circuits for Regular Expression on FPGA.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15, 12 (2007), 1303–1310. https://doi.org/10.1109/TVLSI.2007.909801

[21] Kuan-Ju Lin, Yi-Hsuan Huang, and Chun-Yuan Lin. 2013. Efficient Parallel Knuth-Morris-Pratt Algorithm for Multi-GPUs with CUDA, Vol. 21. https://doi.org/10.1007/978-3-642-35473-1_54

[22] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. 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* (Lausanne, Switzerland) (*ASPLOS '20*). Association for Computing Machinery, New York, NY, USA, 251–265. https://doi.org/10.1145/3373376.3378471

[23] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1, Article 27 (mar 2023), 27 pages. https://doi.org/10.1145/3579453

[24] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. 2010. A regular expression matching using non-deterministic finite automaton. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 73–76. https://doi.org/10.1109/MEMCOD.2010.5558621

[25] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP?. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) (*ICS '17*). Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3079079.3079100

[26] NVidia. 2021. NSight Compute, retrieved April 30, 2024. https://developer.nvidia.com/nsight-compute.

[27] NVidia. 2023. NVCC, retrieved April 30, 2024. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[28] NVidia. 2024. CUDA C++ Programming Guide, retrieved April 30, 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[29] OISF. 2024. Suricata, retrieved April 30, 2024. https://suricata.io/.

[30] OpenAI. 2022. chatGPT, retrieved April 30, 2024. https://chatgpt.com/.

[31] Derek Pao. 2009. A NFA-based programmable regular expression match engine. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Princeton, New Jersey) (*ANCS '09*). Association for Computing Machinery, New York, NY, USA, 60–61. https://doi.org/10.1145/1882486.1882499

[32] PhilipHazel. 2023. PCRE - Perl Compatible Regular Express, retrieved April 30, 2024. https://www.pcre.org/.

[33] Kun Qiu, Harry Chang, Yang Hong, Wenjun Zhu, Xiang Wang, and Baoqian Li. 2021. Teddy: An Efficient SIMD-Based Literal Matching Engine for Scalable Deep Packet Inspection. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) (*ICPP '21*). Association for Computing Machinery, New York, NY, USA, Article 62, 11 pages. https://doi.org/10.1145/3472456.3473512

[34] M. O. Rabin and D. Scott. 1959. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, 2 (1959), 114–125. https://doi.org/10.1147/rd.32.0114

[35] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Pro-*

ceedings of the 13th USENIX Conference on System Administration (Seattle, Washington) (*LISA '99*). USENIX Association, USA, 229–238.

[36] R. Sidhu and V.K. Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. 227–238.

[37] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) (*SIGMOD '17*). Association for Computing Machinery, New York, NY, USA, 403–415. https://doi.org/10.1145/3035918.3035954

[38] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. 2016. SIMD-accelerated regular expression matching. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) (*DaMoN '16*). Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2933349.2933357

[39] Jack Wadden. 2017. Snort inputs, retrieved April 30, 2024. https://github.com/jackwadden/ANMLZoo/tree/master/Snort/inputs.

[40] Jack Wadden. 2017. Snort Regex Patterns, retrieved April 30, 2024. https://github.com/jackwadden/ANMLZoo/blob/master/Snort/regex/snort.1chip.regex.

[41] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) (*NSDI'19*). USENIX Association, USA, 631–648.

[42] Hao Xu, Harry Chang, Wenjun Zhu, Yang Hong, Geoff Langdale, Kun Qiu, and Jin Zhao. 2023. Harry: A Scalable SIMD-based Multi-literal Pattern Matching Engine for Deep Packet Inspection. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, New York City, NY, USA, May 17-20, 2023*. IEEE, 1–10. https://doi.org/10.1109/INFOCOM53939.2023.10229022

[43] Yi-Hua Yang and Viktor Prasanna. 2012. High-Performance and Compact Architecture for Regular Expression Matching on FPGA. *IEEE Trans. Comput.* 61, 7 (2012), 1013–1025. https://doi.org/10.1109/TC.2011.129

[44] Xiaodong Yu and Michela Becchi. 2013. GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) (*CF '13*). Association for Computing Machinery, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/2482767.2482791

[45] Xiaodong Yu and Michela Becchi. 2013. Optimized iNFAnt slides, retrieved April 30, 2024. https://pdfs.semanticscholar.org/31f8/b3fc764e6cf98cf3cbd0aa43ef10e41828b6.pdf.

[46] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. *SIGPLAN Not.* 47, 8 (feb 2012), 129–140. https://doi.org/10.1145/2370036.2145833