HybridSA: GPU Acceleration of Multi-pattern Regex Matching using Bit Parallelism

ALEXIS LE GLAUNEC, Rice University, USA
LINGKUN KONG, Rice University, USA
KONSTANTINOS MAMOURAS, Rice University, USA

Multi-pattern matching is widely used in modern software for applications requiring high throughput such as protein search, network traffic inspection, virus or spam detection. Graphics Processor Units (GPUs) excel at executing massively parallel workloads. Regular expression (regex) matching is typically performed by simulating the execution of deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). The natural implementations of these automata simulation algorithms on GPUs are highly inefficient because they give rise to irregular memory access patterns.

This paper presents HybridSA, a heterogeneous CPU-GPU parallel engine for multi-pattern matching. HybridSA uses bit parallelism to efficiently simulate NFAs on GPUs, thus reducing the number of memory accesses and increasing the throughput. Our bit-parallel algorithms extend the classical shift-and algorithm for string matching to a large class of regular expressions and reduce automata simulation to a small number of bitwise operations. We have developed a compiler to translate regular expressions into bit masks, perform optimizations, and choose the best algorithms to run on the GPU. The majority of the regular expressions are accelerated on the GPU, while the patterns that exhibit random memory accesses are executed on the CPU in parallel. We evaluate HybridSA against state-of-the-art CPU and GPU engines, as well as a hybrid combination of the two. HybridSA achieves between 4 and 60 times higher throughput than the state-of-the-art CPU engine and between 4 and 233 times better than the state-of-the-art GPU engine across a collection of real-world benchmarks.

CCS Concepts: • **Theory of computation** → **Formal languages and automata theory**; **Regular languages**; • **Software and its engineering** → *Semantics.*

Additional Key Words and Phrases: regular expressions, regex matching, CUDA, shift-and algorithm, bit parallelism

ACM Reference Format:
Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2024. HybridSA: GPU Acceleration of Multi-pattern Regex Matching using Bit Parallelism. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 331 (October 2024), 30 pages. https://doi.org/10.1145/3689771

1 Introduction

Regular expressions [Kleene 1956] and finite-state automata (e.g., DFAs and NFAs) describe patterns over sequences and have found applications in numerous domains. They have been used for the lexical analysis of programs [Johnson et al. 1968] during compilation, the search of words and patterns in text editors [Thompson 1968], and bibliographic search [Aho and Corasick 1975]. Regular patterns are also used in network security [Yu et al. 2006] to search for intrusion signatures

Authors' Contact Information: Alexis Le Glaunec, Rice University, Houston, USA, alexis.leglaunec@rice.edu; Lingkun Kong, Rice University, Houston, USA, klk@rice.edu; Konstantinos Mamouras, Rice University, Houston, USA, mamouras@rice.edu.

ACM 2475-1421/2024/10-ART331
https://doi.org/10.1145/3689771

in network traffic, in bioinformatics [Roy and Aluru 2016] for describing protein, RNA, or DNA sequences, and in runtime verification [Bartocci et al. 2018] for specifying safety properties.

Due to the broad applicability of regular patterns, many software tools exist for solving the regular pattern matching problem. These tools are called *regex engines*. Many software implementations are meant to run on CPUs, including the widely-used PCRE [The PCRE2 Developers 2024], grep [Grep 2022], RE2 [RE2 2023], and Hyperscan [Wang et al. 2019]. Given the widespread adoption of dedicated general-purpose GPUs, it is notable that there are only a few efforts for implementing regex matching on GPUs [Avalle et al. 2016; Cascarano et al. 2010; Liu et al. 2020; Yu and Becchi 2013; Zu et al. 2012]. One of the main obstacles in taking advantage of GPUs for regular pattern matching is that the typical algorithms for automata simulation (DFA- or NFA-based) involve highly irregular memory access patterns when reading the transition function/relation from memory. These memory access patterns are not a good fit for GPUs, which require very specific and regular memory accesses to operate efficiently.

In this work, we show that it is possible to leverage the massive parallelism of modern GPUs for multi-pattern regex matching by partitioning the set of patterns into two subsets: (1) those that will execute on the CPU and (2) those that will execute on the GPU. The main idea is that for regular expressions of certain simpler forms, it is possible to implement extremely efficient NFA simulation algorithms that are a great fit for GPUs. This partitioning enables much more effective use of the available hardware, namely the CPU and the GPU. By not trying to use the GPU for patterns that are inherently difficult for the GPU execution model, we can effectively balance the load between the CPU and the GPU and achieve a substantial performance improvement.

Contributions. In this paper, we make the following main contributions:

(1) We show that it is possible to implement efficient variants of the classical shift-and algorithm [Baeza-Yates and Gonnet 1992] for string searching on GPUs. The algorithms that we consider can be used to parallelize and accelerate multi-pattern matching for a significant portion of regular patterns that arise in practice. Our algorithms use *bit parallelism* to perform m NFA transitions with a constant number of logical/arithmetic operations, where m is the width of the GPU registers.
(2) We implement a *compiler* that takes as input a set of regular expressions, identifies the subset that can efficiently execute on the GPU, and then chooses a set of specialized GPU kernels for them to maximize the performance. The compiler performs pattern rewriting optimizations and produces for each kernel a compact binary representation of the NFAs that it will execute. We call this representation an *NFA database*. During execution, the NFA database is loaded into the global memory of the GPU, and the massively parallel execution of the NFAs is initiated.
(3) We perform an experimental evaluation of our implementation for multi-pattern matching by comparing it against the state-of-the-art regex engines Hyperscan [Wang et al. 2019] (CPU-based) and HotStart [Liu et al. 2020] (GPU-based). Our results show that our hybrid CPU-GPU approach is around 4 to 233 times faster than other tools.

The use of bit parallelism in NFA simulation (using arithmetic and bitwise operations) has been explored in several prior works (see, e.g., [Baeza-Yates and Gonnet 1992], [Wu and Manber 1992], [Navarro 2001], [Navarro and Raffinot 2002], and [Wang et al. 2019]). The main *novelties* in this paper concern (i) the careful GPU implementation of bit parallelism to altogether avoid thread divergence and uncoalesced memory accesses and (ii) the use of rewriting optimizations that enable the use of faster GPU kernels.

Paper Outline. Section 2 provides an overview of the necessary definitions and an exposition of NFA simulation using bit vectors for representing sets of states. In Section 3, we describe the GPU implementation (in the CUDA programming model) of several variants of the shift-and algorithm for bit-parallel NFA execution. Section 4 describes the partitioning of the pattern set between the GPU and the CPU, how the GPU kernels are chosen, and some optimizing transformations that can speed up NFA execution on the GPU. In Section 5, we compare our tool (called HybridSA) with Hyperscan [Wang et al. 2019] and HotStart [Liu et al. 2020]. Section 6 contains a discussion of related work. We conclude in Section 7 with a brief summary of our main contributions.

2 Preliminaries

In this section, we give a brief overview of concepts that we will need in the later development: regular expressions, homogeneous NFAs, and the Glushkov construction. We also give a description of the algorithm for simulating the execution of homogeneous NFAs using bit vectors to represent sets of active NFA states.

Let Σ be a finite alphabet. A ***regular expression*** (or *regex*) over Σ is given by the grammar

$$
\begin{aligned}
r ::= {} & \varepsilon \mid && \text{[empty string]} \\
& \sigma \mid && \text{[character class]} \\
& r \cdot r \mid && \text{[concatenation]} \\
& r|r \mid && \text{[nondeterministic choice]} \\
& r^* \mid && \text{[Kleene star]} \\
& r\{m,n\} && \text{[bounded repetition]}
\end{aligned}
$$

where $\sigma \subseteq \Sigma$ is a predicate over the alphabet and m, n are natural numbers with $m \leq n$. The expression $r\{m,n\}$ is called a *bounded repetition* and describes the repetition of r from m to n times. We write $r\{n\}$ for $r\{n,n\}$. The concatenation symbol is sometimes omitted, i.e., we write $r_1 r_2$ instead of $r_1 \cdot r_2$. The *interpretation* of a regex r is a language $\mathcal{L}(r) \subseteq \Sigma^*$, which is defined in the standard way.

A regular expression can be transformed into an equivalent nondeterministic finite automaton (NFA). We will use Glushkov's construction [Glushkov 1961] to convert a regular expression into an NFA. In contrast to Thompson's construction [Thompson 1968], Glushkov's construction results in ε-free automata that are also *homogeneous*, i.e., all incoming transitions of a state are labeled with the same predicate over the alphabet.

Definition 1 (Homogeneous NFA). Let Σ be a finite alphabet. A *homogeneous nondeterministic finite automaton* with input alphabet Σ is a tuple $\mathcal{A} = (Q, L, \Delta, I, F)$, where

- Q is a finite set of *(control) states*,
- $L : Q \to \mathcal{P}(\Sigma)$ is a labeling function that maps each state to a character class,
- $\Delta : Q \to \mathcal{P}(Q)$ is the *transition relation*,
- $I \subseteq Q$ is the set of *initial states*, and
- $F \subseteq Q$ is the set of *final states*,

where $\mathcal{P}$ is the powerset operation.

Given a set of regular expressions and an input string, the multi-pattern matching problem involves simulating a set of regular expressions over the same input text and reporting the total number of matches. Multi-pattern matching is an embarrassingly parallel problem as the NFAs can be executed independently over many independent inputs. A standard approach to solve the multi-pattern matching problem is to execute each NFA independently. This is shown in Algorithm 1, where a Boolean transition matrix is used to represent the transitions of the automaton.

Algorithm 1: Simulation of the execution of a homogeneous NFA

```
1 Kernel NfaSimulation(init, final, labels, delta, text):
      // delta is the Boolean transition matrix of the NFA
      // i.e., delta[i][j] = 1 iff there is a transition from state i to state j
2     states ← 0 0 ... 0 0   // n-bit vector of active states, where n is the # NFA states
3     nMatches ← 0   // number of matches
4     for c in text do // left-to-right pass over input text
5         next ← init   // init states are always active
6         next ← next OR (states · delta)   // execute transition by matrix multiplication
7         states ← next AND labels[c]   // keep active states with the right label
8         nMatches ← nMatches + ((states AND final) ≠ 0 0 ... 0 0)
9     return nMatches
```

Notation (Bit Vectors). We will use the notation $x_{n-1}\,x_{n-2}\,\ldots\,x_2\,x_1\,x_0$ for an n-bit vector. The i-th position of the bit vector has the value x_i. If we think of the bit vector as a binary representation of a number (e.g., a GPU register), the leftmost bit is the most significant and the rightmost bit is the least significant. The *left shift* operation $\ll$ shifts each bit to the left and inserts 0 in the least significant position. For example, $x_3\,x_2\,x_1\,x_0 \ll 1 = x_2\,x_1\,x_0\,0$ and $x_3\,x_2\,x_1\,x_0 \ll 2 = x_1\,x_0\,0\,0$.

We sometimes use the term *(bit) mask* to refer to a bit vector. This is because they are commonly used as OR masks (to set specific bits to 1) or as AND masks (to set specific bits to 0).

If v is a $1 \times n$ matrix (i.e., a row vector) and M is an $n \times n$ matrix, then their product $v \cdot M$ is a $1 \times n$ matrix, given by $(v \cdot M)(j) = \sum_{i=0}^{n-1} v(i) \cdot M(i,j)$ for every $j = 0, 1, \ldots, n-1$. We will use this operation of *vector-matrix multiplication* for the product of a state bit vector with the Boolean transition matrix of an NFA.

The arguments *init*, *final*, *labels* and *delta* of `NfaSimulation` in Algorithm 1 represent a homogeneous NFA $\mathcal{A} = (Q, L, \Delta, I, F)$ with $n = |Q|$ states. Assume w.l.o.g. that $Q = \{0, 1, \ldots, n-1\}$. Both *init* and *final* are n-bit vectors that represent the sets I and F respectively. E.g., for every $q \in Q$, we have that $init(q) = 1$ iff $q \in I$. The argument *labels* is an array of n-bit vectors that represents L as follows: $labels[c](q) = 1$ iff $c \in L(q)$, for every $c \in \Sigma$ and $q \in Q$. Finally, *delta* is an $n \times n$ matrix that satisfies: $delta(q, q') = 1$ iff $q' \in \Delta(q)$, for all $q, q' \in Q$.

Example 2. Fig. 1 presents the execution of Algorithm 1 for the regex r_1 = `[ab](c|b.*c)` over the input stream abc. Notice that we label the states (instead of the edges) with character classes, as the automaton is homogeneous. The character masks for `a`, `b` and `c` are `01001`, `01011`, and `11100` respectively. For all other letters, the character mask is `01000` as state q_3 is labeled with `.`, which matches any character. The mask for the set of initial states $\{q_0\}$ (resp., set of final states $\{q_2, q_4\}$) is `00001` (resp., `10100`). After reading the character `a`, we perform vector-matrix multiplication (line 6) using the transition matrix *delta*, which gives

$$next = init \text{ OR } (states \cdot delta) = 00001 \text{ OR } (00000 \cdot delta) = 00001.$$

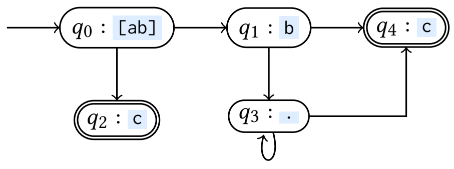
Then, we compute *states*, the set of states activated that also match the current input letter with

$$states = next \text{ AND } labels[\texttt{a}] = (00001 \text{ AND } 01001) = 00001.$$

This means that state q_0 is active. We compute the intersection between *states* and *final* in

$$is\_final = (states \text{ AND } final) \neq 00000 = (00001 \text{ AND } 10100) \neq 00000 = \text{false}.$$

Thus, we have no match. In the next round, we consume character `b`. Right after line 6, we have that $next = (00001 \text{ OR } (00001 \cdot delta)) = 00111$. This is because the successors of state q_0 are states



(a)

Input	a	b	c
next	00001	00111	11111
labels	01001	01011	11100
states	00001	00011	11100
nMatches	0	0	1

(b)

Fig. 1. NFA execution for the pattern r_1 = `[ab](c|b.*c)` over the input string `abc`.

q_1 and q_2. Then, we have that *states* = (00111 AND 01011) = 00011, corresponding to the set of active states $\{q_0, q_1\}$ that both match letter b. Since *is_final* = ((00011 AND 10100) ≠ 00000) = false, we still have no match. For the last character `c`, *next* = (00001 OR(00011 ·*delta*)) = 11111, which means that all states are active in *next*. We compute *states* = (11111 AND 11100) = 11100 and obtain that *is_final* = ((11100 AND 10100) ≠ 00000) = true, corresponding to a match for state q_4 which is final. Thus, we report a match and increment *nMatches*.

For a regular expression r over Σ, a string $w \in \Sigma^*$, and a position $j = 0, 1, \ldots, |w|$, we say that r has a *match* at position j in w if the substring $w[i..j]$ is in $\mathcal{L}(r)$ for some $i = 0, \ldots, j$. We define the function $[\![r]\!] : \Sigma^* \to \mathbb{N}$ as follows: $[\![r]\!](w)$ is equal to the size (i.e., cardinality) of the set

$$\{j \in \mathbb{N} \mid \text{there is } i \text{ s.t. } 0 \le i \le j \le |w| \text{ and } w[i..j] \in \mathcal{L}(r)\}.$$

In other words, $[\![r]\!](w)$ is the number of positions j for which there is a match for r whose right endpoint is j (the left endpoint can be any position $i \le j$).

Problem 3 (Multi-Pattern Matching). Let Σ be a finite alphabet, $r_0, r_1, \ldots, r_{k-1}$ be a sequence of k regular expressions over Σ, and w be a finite string over Σ. For every $i = 0, 1, \ldots, k-1$ compute the number $[\![r_i]\!](w)$ of matches for the pattern r_i.

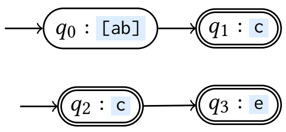
Notice that we are considering a computational problem that is more general than purely Boolean pattern matching. We choose this problem in order to have a more direct comparison with the GPU engine HotStart (see Section 5).

3 Overview of HybridSA

HybridSA uses bit parallelism to simulate NFA transitions more effectively by taking advantage of the simple matrix structure of the NFAs for real-world regexes. In this section, we present four families of kernels used in HybridSA to accelerate regexes on the GPU: ShiftAnd, ShiftAndGap, ShiftAndDist, and ShiftAndOps. We also present a solution to the multi-pattern matching problem (which is embarrassingly parallel) using the CUDA programming model.

3.1 Shift-And Algorithm

Shift-And [Baeza-Yates and Gonnet 1992] is a bit-parallel algorithm that uses bitwise operations to simulate the execution of automata. It consists of two steps: (1) preprocessing of the pattern to derive bit masks that encode the NFA for the pattern, and (2) matching over the input text. An important restriction is that the pattern length must be smaller than the register size (32 bits on the GPU). Shift-And can be viewed as the simulation of a homogeneous automaton, where the states can be placed in order $q_0, q_1, \ldots, q_{n-1}$ on a line and each transition is from a state q_i to a neighboring state q_{i+1}. This transition pattern does not require the matrix multiplication operation seen in Algorithm 1. In fact, matrix multiplication can be replaced by a bitwise shift operation. Compared



(a)

Input	a	c	e
next	0101	0111	1101
labels	0001	0110	1000
states	0001	0110	1000
nMatches	0	1	2

(b)

Fig. 2. Shift-And execution for the pattern r = `[ab]c|ce?` over the input string `ace`.

to Algorithm 1, the Shift-And algorithm can only encode transitions of the form $(i) \longrightarrow (i+1)$. In particular, it cannot simulate back edges, i.e., transitions of the form $(i) \longrightarrow (j)$ where $j \leq i$.

Example 4. Consider the regex r = `[ab]c|ce?`. The automaton $\mathcal{A}_1$ of Fig. 2a recognizes the language of r. The main idea of the Shift-And algorithm is to execute all the transitions $(i) \longrightarrow (i+1)$ with arithmetic and bitwise operations. The initial (resp. final) mask is *maskInitial* = 0101 (resp. *maskFinal* = 1110) and encodes the positions of the initial (resp. final) states. After consuming the character `a`, we start by performing the transitions with

$$\mathit{next} = (\mathit{states} \ll 1) \text{ OR } \mathit{maskInitial} = (0000 \ll 1) \text{ OR } 0101 = 0101.$$

Compared to Example 2, we have replaced matrix multiplication with a shift left operation. Then, we compute the active states that match letter `a` with

$$\mathit{states} = \mathit{next} \text{ AND } \mathit{labels}[\texttt{a}] = (0101 \text{ AND } 0001) = 0001.$$

This means that the initial state q_0 is active. To check if there is a match, we compute the Boolean test ((*states* AND *maskFinal*) $\neq$ 0000), which is equal to false. So, there is no match.

After consuming letter `c`, we have *next* = (0001 $\ll$ 1) OR 0101 = 0111. We compute *states* = (0111 AND 0110) = 0110. As *is_final* = ((0110 AND 1110) $\neq$ 0000) = true, we report a match. Finally, after consuming letter `e`, we have *next* = (0110 $\ll$ 1) OR 0101 = 1101, *states* = (1101 AND 1000) = 1000, and *is_final* = ((1000 AND 1110) $\neq$ 0000) = true.

3.2 Shift-And on a GPU

CUDA devices consist of single-instruction, multiple threads (SIMT) processors called *Streaming Multiprocessors* (34 on RTX 4060Ti). Each multiprocessor has a limited number of 32-bit *registers* (65 K registers per SM), *shared memory* that is local to a multiprocessor (100 KB on RTX 4060Ti) and *global memory* that is shared among the multiprocessors (8 GB on RTX 4060Ti). CUDA threads are executed on a multiprocessor, 32 threads at a time, in a CUDA *warp*. Within a warp, all 32 threads execute the same instruction at every step. CUDA warps are grouped in CUDA *thread blocks*, which are scheduled to the device's multiprocessors and executed in parallel. To execute code on the GPU, programmers implement functions called CUDA *kernels* using a variant of C++. Each kernel execution is parameterized by (1) the number of CUDA thread blocks to execute in parallel, and (2) the number of threads per CUDA thread block. The organization of all threads of a kernel in blocks is referred to as the thread *grid*. All threads created during kernel invocation run the same piece of code. They are differentiated by using the special variables `blockIdx`, which specifies the block identifier of the current thread, and `threadIdx`, which specifies the current thread identifier within its block. More information about the CUDA programming model can be found in [NVidia 2024].

We propose an efficient CUDA implementation of the Shift-And algorithm for the embarrassingly parallel multi-pattern matching problem. In our algorithm, each CUDA thread simulates the

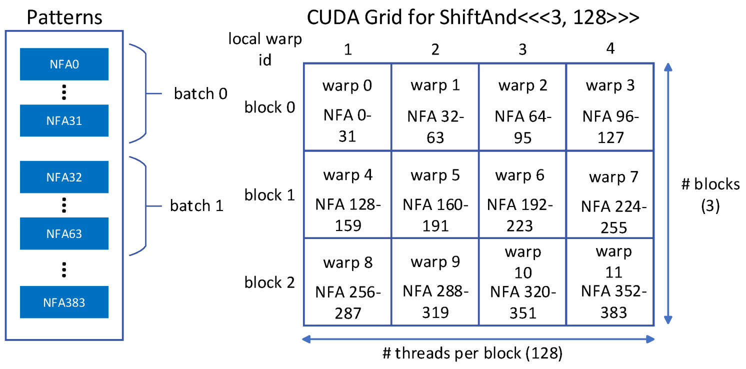


Fig. 3. Layout of the NFAs on the GPU for a grid of size (3,128).

execution of one NFA. This means that a warp, which is a collection of 32 threads, executes a group of 32 NFAs, which we call a *batch*. Fig. 3 shows the layout of the Shift-And NFA batches in the CUDA grid. Assuming that our input pattern set consists of n regexes, each regex and its corresponding Glushkov NFA is uniquely determined by a number i in the range $0, 1, \ldots, n-1$. We call this the *global NFA identifier* of the automaton. The n NFAs are grouped into $\lceil n/$ `N_NFAS_PER_BATCH` $\rceil$ batches, where `N_NFAS_PER_BATCH` $= 32$. The NFA with global identifier i is placed in the batch with identifier $\lfloor i/$ `N_NFAS_PER_BATCH` $\rfloor$ and it has a batch-local identifier $j = i \bmod$ `N_NFAS_PER_BATCH`, which we call a *local NFA identifier*. The data that describes the NFAs of a batch is organized in the structure `Batch_SA` (see below). For $T = u32$ (type of GPU registers), the size of the `Batch_SA` structure is equal to $2\cdot$ `N_NFAS_PER_BATCH` $\cdot \text{sizeof}(u32) + 256\cdot$ `N_NFAS_PER_BATCH` $\cdot \text{sizeof}(u32) = 2\cdot 32\cdot 4 + 256\cdot 32\cdot 4 \approx 32\,\text{KB}$ stored in global memory for a given batch. For each NFA, this structure contains the mask of initial states, the mask of final states, and the mask for each character of the input alphabet. The data in `Batch_SA` uses a carefully chosen layout to ensure all thread accesses in a warp are coalesced.

```
template<typename T> struct Batch_SA {
    T masks_initial[N_NFAS_PER_BATCH]; // masks for initial states
    T masks_final[N_NFAS_PER_BATCH]; // masks for final states
    T masks_char[N_CHARS][N_NFAS_PER_BATCH]; // masks for states that match each character
};
```

More precisely, the initial and final masks are 1D arrays indexed by a local NFA id (index 0 to 31) for a given batch. The character mask is a 2D array stored in a character-first order. The purpose of this layout is to have coalesced accesses when reading the character mask for the current input character. All `Batch_SA` arrays are parameterized by a type T that specifies the data type used to encode sets of states (it can be something other than 32-bit, the register size of the GPU). Thus, Shift-And can be extended to any data type implementing the shift left operation ($\ll$), bitwise or (OR), bitwise and (AND), and the equality operator ($=$).

Algorithm 2 presents pseudocode for the Shift-And kernel. We will use the example of a problem instance with 384 NFAs to explain the algorithm. We will use 384 CUDA threads to execute the NFAs in parallel. These threads will be organized in a thread grid with $M = 3$ CUDA blocks and $N = 128$ threads per block. This means that there will be 4 warps per block. Notice that $M \cdot N = 384$. We invoke the kernel with `ShiftAnd<<<M,N>>>`. The parameters inside the angled brackets define the CUDA grid dimension which is of size $M \times N = 384$ in this example. Fig. 3 shows the mapping of the NFAs onto the CUDA grid. A batch (32 NFAs) is executed by a CUDA warp (32 threads) and a

Algorithm 2: Shift-And on the GPU

```
1  Kernel ShiftAnd<T>($output, batches, text$):
       // Each thread of the grid handles the execution of exactly one NFA
       // $blockId$ is the block identifier in the current grid (blockIdx.x in CUDA)
       // $threadId$ is the thread identifier within the current block (threadIdx.x in CUDA)
       // N_THREADS_PER_WARP = 32 is a constant of the CUDA programming model
2      $localWarpId \leftarrow threadId \div$ N_THREADS_PER_WARP   // warp id within the block
       // $nThreadsPerBlock$ is the number of threads per block (blockDim.x in CUDA)
3      assert $nThreadsPerBlock$ mod N_THREADS_PER_WARP = 0
4      $nWarpsPerBlock \leftarrow nThreadsPerBlock \div$ N_THREADS_PER_WARP   // # warps in each block
5      $globalWarpId \leftarrow blockId \cdot nWarpsPerBlock + localWarpId$   // warp id within the grid
       // Each warp of the grid handles exactly one batch of NFAs
       // Each batch contains N_THREADS_PER_WARP NFAs
6      $batchId \leftarrow globalWarpId$   // identifier for the batch to be processed
7      &BatchSA<T> $batch \leftarrow \&batches[batchId]$   // pointer to batch
8      $localNfaId \leftarrow threadId$ mod N_THREADS_PER_WARP   // NFA id within the warp/batch
9      $T$ $maskInitial \leftarrow batch$.masks_initial[$localNfaId$]   // mask of initial states
10     $T$ $maskFinal \leftarrow batch$.masks_final[$localNfaId$]   // mask of final states
       // Execution of homogeneous NFA:
11     $T$ $states \leftarrow 0$ // set of active states
12     $nMatches \leftarrow 0$ // number of matches
13     for $c$ in $text$ do // left-to-right pass over input text
14         $mask \leftarrow batch$.masks_char[$c$][$localNfaId$]   // get mask for character $c$
15         $states \leftarrow ((states \ll 1)$ OR $maskInitial)$ AND $mask$   // execute transition
16         $nMatches \leftarrow nMatches + ((states$ AND $maskFinal) \neq 0)$
17     $globalNfaId \leftarrow batchId \cdot$ N_NFAS_PER_BATCH $+ localNfaId$
18     assert $globalNfaId = blockId \cdot nThreadsPerBlock + threadId$
19     $output[globalNfaId] \leftarrow nMatches$
```

block corresponds to 4 warps for a total of 384 NFAs in the dataset. There are 128 threads per block or, equivalently, 128 NFAs per block.

For example, the NFA with global identifier 200 belongs to block 1 and is executed by the warp with global id 6. At runtime, the CUDA scheduler dispatches blocks to multiprocessors (SM) and the SM scheduler selects a warp to be executed. Suppose that block 1 is dispatched to SM0 which executes warp 6 that contains NFA 200. In Algorithm 2, the execution of NFA 200 (in parallel with all the NFAs of warp 6) starts by retrieving *blockId*, the block identifier in the grid, and *threadId*, the thread identifier within the block. For NFA 200, we have $blockId = 1$ and $threadId = 72$ as NFA 200 belongs to block 1 and is the 73rd NFA in the block (out of 128). To retrieve the Shift-And masks, which are stored per batch, we need to calculate the global batch id and the local NFA id. First, we calculate the local warp id (within the block) as follows: $localWarpId = threadId \div$ N_THREADS_PER_WARP $= 72 \div 32 = 2$. Then, we calculate the global warp id (within the grid) by $globalWarpId = blockId \cdot nWarpsPerBlock + localWarpId = 1 \cdot 4 + 2 = 6$. Since one batch corresponds exactly to one warp, we also obtain that $batchId = 6$. We calculate the local NFA id by $localNfaId = threadId$ mod N_THREADS_PER_WARP $= 72$ mod $32 = 8$. Now, we can obtain the masks *maskInitial* and *maskFinal* by indexing with the local NFA id.

From Line 11, the algorithm starts with the simulation of the execution of the NFA. For every character c of the input text, we fetch the character mask by first indexing using c and then using the local NFA id. We continue to simulate the transition of the NFA using the Shift-And algorithm. This requires only *1 left shift, 1 bitwise OR, and 1 bitwise AND*. We continue to check whether the pattern matches at the current location by comparing the active states with the final states (if they overlap, then there is a match). At the end of the execution, we calculate the global NFA id (within the entire pattern set) as follows: $globalNfaId = batchId \cdot$ `N_NFAS_PER_BATCH` $+ localNfaId = 6 \cdot 32 + 8 = 200$ and we store the number of matches for NFA 200 in the output array.

In Algorithm 2, the layout and access pattern of the `Batch_SA` structure are carefully chosen to reduce the latency incurred by accesses to the off-chip global memory. In fact, we make sure that all the threads within a warp make contiguous memory accesses, generally referred to as ***coalesced accesses***. In the Shift-And pseudocode, the 32 threads of a warp access the same `Batch_SA` structure (*batchId* is computed directly from the *blockId* which is identical for the threads in a warp). Then, the 1-dimensional mask arrays *maskInitial* and *maskFinal* are indexed with *localNfaId*, which takes values from 0 to 31. The 2-dimensional array masks_char is accessed with two indexes: a character c and an index *localNfaId*. For a fixed character c, masks_char$[c]$ is a row of the 2D array and is stored in consecutive memory locations. For this reason, the memory accesses masks_char$[c][localNfaId]$ in a warp are coalesced. More details about memory coalescing can be found in [NVidia 2013a]. Algorithm 2 does not contain any branching that could possibly make the threads of a warp diverge. ***Thread divergence*** occurs when at least two threads disagree on the execution path within a warp and can be caused by a data-dependent conditional branch. In the event of a thread divergence, the warp executes each branch path taken serially, and the threads that are not on that path are disabled. Algorithm 2 does not use any shared memory [NVidia 2013b], which is a type of memory that can be used for inter-thread communication within a warp (in fact, within a thread block), and that often requires ***warp-level synchronization*** leading to additional complexity.

3.3 Extending Shift-And

The Shift-And algorithm only supports regular expressions that can be rewritten as a union of strings, which accounts for roughly 20% of all regexes over the datasets of Section 5. We would like to accelerate a larger percentage of the regular expressions on the GPU. Therefore, we propose extensions to the Shift-And algorithm to support more classes of regular expressions on the GPU.

ShiftAndDist. Shift-And restricts transitions from a state (i) to the next state $(i+1)$, but many regular expressions require transitions that step over some states. We also need to support self-loops, i.e., transitions from state (i) to itself. To support more NFAs, we can generalize the bit-parallel simulation of transitions to distances other than 1. This is the main idea of the ShiftAndDist kernel. We group together NFAs with transition distance bounded above by a constant D. Algorithm 3 presents the CUDA kernel for the ShiftAndDist algorithm. The kernel is parameterized with the datatype T and the maximum distance D to execute transitions of distances from 0 to D (inclusive). Therefore, we can support all transitions of the form $(i) \longrightarrow (i+d)$ with $0 \leq d \leq D$, which we call shift transitions. The number of operations needed for ShiftAndDist is proportional to the maximum transition distance D. We use the structure `Batch_SA_Dist` for representing the NFAs that are handled by ShiftAndDist. Compared to `Batch_SA`, we add the 2-dimensional mask array masks_dist, where masks_dist$[d]$ contains the source states of transitions of distance d. All threads within a warp always access the same memory location in *maskPerDist* (see lines 10 and 12), which ensures coalesced memory accesses. Moreover, ShiftAndDist completely avoids thread divergence.

```
template<typename T, uint8_t D> struct Batch_SA_Dist {
    T masks_initial[N_NFAS_PER_BATCH]; // masks for initial states
    T masks_final[N_NFAS_PER_BATCH]; // masks for final states
    T masks_dist[D + 1][N_NFAS_PER_BATCH]; // masks for source states of edges with a given length
    T masks_char[N_CHARS][N_NFAS_PER_BATCH]; // masks for states that match each character
};
```

Algorithm 3: ShiftAndDist for distance D on the GPU

```
1  Kernel ShiftAndDist<T, D>(output, batches, text):
2      ...  // compute batchId and localNfaId as in ShiftAnd
3      &BatchSADist<T, D> batch ← &batches[batchId]   // pointer to batch
4      T maskInitial ← batch.masks_initial[localNfaId]   // mask of initial states
5      T maskFinal ← batch.masks_final[localNfaId]   // mask of final states
6      T[] maskPerDist ← batch.masks_dist   // pointer to masks for each distance ≤ D
7      T states ← 0 // set of active states
8      nMatches ← 0 // number of matches
9      for c in text do // left-to-right pass over input text
10         next ← states AND maskPerDist[0]   // states with a self-loop
11         for d = 1, ..., D do
12             next ← next OR ((states AND maskPerDist[d]) ≪ d)
13         mask ← batch.masks_char[c][localNfaId]   // get mask for character c
14         states ← (next OR maskInitial) AND mask   // execute transition
15         nMatches ← nMatches + ((states AND maskFinal) ≠ 0)
16     globalNfaId ← batchId · N_NFAS_PER_BATCH + localNfaId
17     output[globalNfaId] ← nMatches
```

Example 5. Consider the regex r = `ab{0,2}c`. The automaton below recognizes the language of r.

$$\begin{tikzcd} {} \arrow[r] & q_0 : \mathtt{a} \arrow[r] \arrow[rrr, dashed, bend right] & q_1 : \mathtt{b} \arrow[r] \arrow[rr, dotted, bend right] & q_2 : \mathtt{b} \arrow[r] & q_3 : \mathtt{c} \end{tikzcd}$$

In this example, the maximum transition distance for the NFA is $D = 3$ and it corresponds to the dashed arrow. Using Shift-And, we are only able to simulate the solid line transitions (distance 1). `ShiftAndDist` adds more expressiveness to our implementation by allowing for transitions with various distances. Here, the loop at line 11 in Algorithm 3 will simulate transitions of distances 0, 1, 2 and 3 (even though there is no transition of distance 0).

ShiftAndGap. We have implemented the specialized algorithm ShiftAndGap for the commonly occurring regexes that contain gap transitions such as $ab\{0,k\}c$ with the transition going from state a to state c. The structure `Batch_SA_Gap` extends the existing `Batch_SA` structure with two additional fields: *maskGapI* and *maskGapF*. The mask *maskGapI* contains the positions of the states right before gaps, and *maskGapF* contains the positions of the states right after gaps. Algorithm 4 presents the pseudocode for ShiftAndGap. To efficiently compute the ε-transitions, we use a bit trick at line 13. For datatype $T = u32$, the size of the `Batch_SA_Gap` is of about $4 \cdot$ `N_NFAS_PER_BATCH` $\cdot \operatorname{sizeof}(u32) + N\_CHARS \cdot$ `N_NFAS_PER_BATCH` $\cdot \operatorname{sizeof}(u32) = 4 \cdot 32 \cdot 4 + 256 \cdot 32 \cdot 4 \approx 33$ KB which is roughly the same as for `Batch_SA`. In general, the batch is dominated by the size of masks_char.

```
template<typename T> struct Batch_SA_Gap {
    T masks_initial[N_NFAS_PER_BATCH]; // masks for initial states
    T masks_final[N_NFAS_PER_BATCH]; // masks for final states
    T masks_gap_initial[N_NFAS_PER_BATCH]; // masks for gap-initial states
    T masks_gap_final[N_NFAS_PER_BATCH]; // masks for gap-final states
    T masks_char[N_CHARS][N_NFAS_PER_BATCH]; // masks for states that match each character
}
```

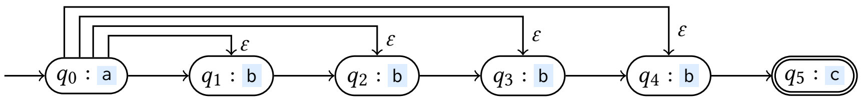
Algorithm 4: ShiftAndGap on the GPU

```
Kernel ShiftAndGap<T>($output$, $batches$, $text$):
    ...  // compute $batchId$ and $localNfaId$ as in ShiftAnd
    &BatchSAGap<T> $batch \leftarrow$ &$batches$[$batchId$]   // pointer to batch
    $T$ $maskInitial \leftarrow batch$.masks_initial[$localNfaId$]   // mask of initial states
    $T$ $maskFinal \leftarrow batch$.masks_final[$localNfaId$]   // mask of final states
    $T$ $maskGapI \leftarrow batch$.masks_gap_initial[$localNfaId$]   // mask of gap-initial states
    $T$ $maskGapF \leftarrow batch$.masks_gap_final[$localNfaId$]   // mask of gap-final states
    $T$ $states \leftarrow 0$ // set of active states
    $nMatches \leftarrow 0$ // number of matches
    for $c$ in $text$ do // left-to-right pass over input text
        $mask \leftarrow batch$.masks_char[$c$][$localNfaId$]   // get mask for character $c$
        $states \leftarrow ((states \ll 1)$ OR $maskInitial)$ AND $mask$   // execute transition on $c$
        // epsilon transitions for gaps:
        $eps \leftarrow (maskGapF - (states$ AND $maskGapI))$ AND (NOT $maskGapF$)
        $states \leftarrow states$ OR $eps$
        $nMatches \leftarrow nMatches + ((states$ AND $maskFinal) \neq 0)$
    $globalNfaId \leftarrow batchId \cdot$ N_NFAS_PER_BATCH $+ localNfaId$
    $output$[$globalNfaId$] $\leftarrow nMatches$
```

Table 1. Execution of the kernel `ShiftAndGap<u32>` for the pattern r = `ab{0,4}c` and the string `abbc`.

Input	`a`	`b`	`b`	`c`
states (before line 11)	000000	011111	011110	011100
states (after line 12)	000001	011110	011100	100000
eps (after line 13)	011111	000000	000000	000000
states (after line 14)	011111	011110	011100	100000
nMatches	0	0	0	1

Example 6. Consider the regex r = `ab{0,4}c`. The following automaton $\mathcal{A}_1$ (nondeterministic with epsilon transitions) recognizes the language of r.



In this example, we have a gap consisting of `b{0,4}`, the repetition of `b` between 0 and 4 times. In the automaton, there is a single gap-initial state, namely q_0, the state right before the gap `b{0,4}`. There is a single gap-final state, namely q_5, the state right after the gap. Table 1 presents the execution of $\mathcal{A}_1$ by kernel `ShiftAndGap<u32>` for the input string `abbc`. The main idea of ShiftAndGap is to execute all the epsilon transitions into the gap with arithmetic and bitwise operations. The values of the masks used by ShiftAndGap for $\mathcal{A}_1$ are the following:

$$maskInitial = 000001 \quad maskFinal = 100000 \quad maskGapI = 000001 \quad maskGapF = 100000$$

The character masks for `a`, `b` and `c` are 000001, 011110, and 100000 respectively. After consuming the character `a`, we start by performing the transitions of distance 1 with

$$\begin{aligned} states &= ((states \ll 1) \text{ OR } maskInitial) \text{ AND } mask \\ &= ((000000 \ll 1) \text{ OR } 000001) \text{ AND } 000001 = 000001. \end{aligned}$$

Then, we compute *eps*, the set of states activated by the epsilon transitions, in three steps. First, we compute the set of gap-initial states that are active with

$$\mathit{states}\ \text{AND}\ \mathit{maskGapI} = \texttt{000001}\ \text{AND}\ \texttt{000001} = \texttt{000001}.$$

Then, we identify the destination states for the epsilon transitions with

$$\begin{aligned}\mathit{eps} &= (\mathit{maskGapF} - (\mathit{states}\ \text{AND}\ \mathit{maskGapI}))\ \text{AND}\ (\text{NOT}\ \mathit{maskGapF})\\ &= (\texttt{100000} - \texttt{000001})\ \text{AND}\ \texttt{011111}\\ &= \texttt{011111}\ \text{AND}\ \texttt{011111},\end{aligned}$$

which is equal to `011111`. Notice that this bit vector includes the source state q_0, but this is not an issue because it is already enabled in *states*. So, at the end of the first round, *states* = `011111`.

Let us consider now the body of the loop when we continue to consume the character `b`. After line 12 executes, we have that

$$\begin{aligned}\mathit{states} &= ((\texttt{011111} \ll 1)\ \text{OR}\ \mathit{maskInitial})\ \text{AND}\ \mathit{mask}\\ &= ((\texttt{011111} \ll 1)\ \text{OR}\ \texttt{000001})\ \text{AND}\ \texttt{011110} = \texttt{011110}.\end{aligned}$$

Then, we proceed to execute line 13, which computes the value

$$\begin{aligned}\mathit{eps} &= (\mathit{maskGapF} - (\mathit{states}\ \text{AND}\ \mathit{maskGapI}))\ \text{AND}\ (\text{NOT}\ \mathit{maskGapF})\\ &= (\texttt{100000} - (\texttt{011110}\ \text{AND}\ \texttt{000001}))\ \text{AND}\ \texttt{011111} = \texttt{000000}.\end{aligned}$$

This means that no epsilon transitions are taken. This is expected, because the epsilon transitions are only taken right after we see a character `a`. So, after line 14, we have that *states* = `011110`.

Suppose that in the third round we consume the character `b` again. After line 12 is executed, we have that *states* = ((`011110` $\ll$ 1) OR `000001`) AND `011110` = `111101` AND `011110` = `011100`. As in the previous round, we get that *eps* = `000000` after executing line 13. It follows that *states* = `011100` after line 14.

In the final round of this example's execution, we read the character `c`. After executing line 12, we get *states* = ((`011100` $\ll$ 1) OR `000001`) AND `100000` = `111001` AND `100000` = `100000`. No epsilon transitions are enabled and therefore *eps* = `000000` after line 13 is executed. So, *states* = `100000` after line 14. This means that the variable *nMatches* will be incremented by 1 at the end of this round, which indicates that a match of the regex has been found.

ShiftAndOps. In theory, `ShiftAndDist` can simulate all NFAs that have transitions of type $(i) \to (j)$ where $i \leq j$. This corresponds to NFAs without back edges. Given an NFA with n states, the transition distance is bounded by $n - 1$. However, in practice this bound can be too large for efficient execution on the GPU. Take as example the regex `[A-Z0-9_-.]{20,300}x3b` from the Suricata dataset. The bound on the transition distance here is 281, thus we would iterate 282 (including distance 0) times over the loop starting at line 11 in Algorithm 3, leading to disastrous performance. Fig. 4 presents the distribution of the maximum transition distance across the Prosite, Snort, SpamAssassin, Suricata and Yara datasets which are described in more detail in Section 5. We observe that the majority of regexes have small maximum transition distance. For all datasets except Prosite, a non-negligible number of regexes have maximum distance larger than 10. Moreover, ShiftAndDist cannot support regexes with back edges, i.e. transitions of the form $(i) \to (j)$ with $j < i$. We introduce a new algorithm to deal efficiently with regexes containing long transitions and to support those with backward transitions. We call it ShiftAndOps.

Algorithm 5 presents the ShiftAndOps kernel, which extends the kinds of NFA transitions that are supported on the GPU by introducing multi-edge transitions.

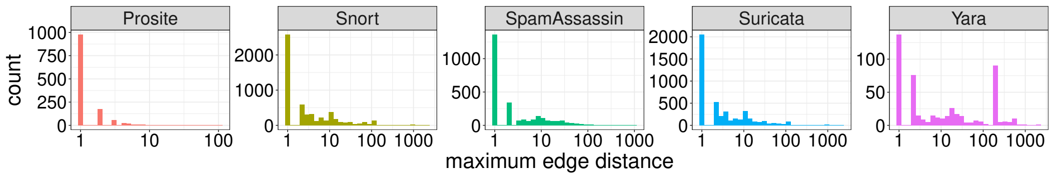
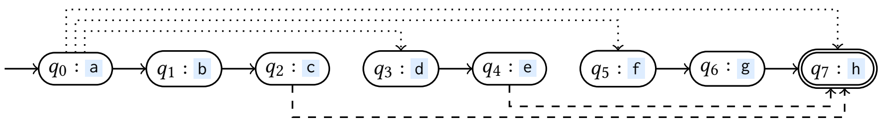


Fig. 4. Distribution of the maximum transition distance across the datasets

We introduce two kinds of *multi-edge transitions* to efficiently execute regexes with disjunctions: one-to-many and many-to-one transitions. A one-to-many transition has the form $(q_i) \longrightarrow (q_{k_1}, q_{k_2}, ...q_{k_n})$ and can be thought of as the collection of the n individual NFA transitions $q_i \longrightarrow q_{k_1}$, $q_i \longrightarrow q_{k_2}$, ..., $q_i \longrightarrow q_{k_n}$. So, if q_i is active, then the states $q_{k_1}, q_{k_2}, \ldots, q_{k_n}$ are activated after the transition is performed. One-to-many transitions arise in regexes with disjunctions such as r = `a(b|c?b|b+)`, where the first state labeled with `a` activates the first states in each disjunction. A many-to-one transition has the form $(q_{k_1}, q_{k_2}, \ldots, q_{k_n}) \longrightarrow (q_i)$ and is essentially the collection of the n individual NFA transitions $q_{k_1} \longrightarrow q_i, q_{k_2} \longrightarrow q_i, \ldots, q_{k_n} \longrightarrow q_i$. If any of the states $q_{k_1}, q_{k_2}, \ldots, q_{k_n}$ is active, then the state q_i gets activated after the transition is performed. Many-to-one transitions are useful for regexes with disjunctions such as r = `(a|b+|cd)e`, where any of the states labeled with `a`, `b` or `d` can activate the state labeled with `e`. Compared to Algorithm 3 for the `ShiftAndDist` kernel, Algorithm 5 adds a new block starting at line 14 to simulate the multi-edge transitions. The structure `Batch_SA_Ops` extends `Batch_SA` with the four additional mask arrays masks_shift, masks_dist, masks_src and masks_dst. The bit vector masks_shift[i][*localNfaId*] contains the source states for the i-th shift operation for NFA *localNfaId* of the batch. The integer shift_dist[i][*localNfaId*] is the distance of the i-th shift operation. The bit vector masks_src[i][*localNfaId*] contains the source states which activate the destination states in masks_dst[i][*localNfaId*] for NFA *localNfaId* of the batch. In the block executing multi-edge transitions (starting at line 14), one-to-many and many-to-one transitions are executed using the same instructions. Example 7 shows the masks used by ShiftAndOps for the execution of regex r = `a(bc|de|fg|)h`.

```
template<typename T, uint8_t M, uint8_t N> struct Batch_SA_Ops {
    T masks_initial[N_NFAS_PER_BATCH]; // masks for initial states
    T masks_final[N_NFAS_PER_BATCH]; // masks for final states
    T masks_shift[M][N_NFAS_PER_BATCH]; // masks for source states of edges with a given length
    uint32_t shift_dist[M][N_NFAS_PER_BATCH]; // distance for each shift operation
    T masks_src[N][N_NFAS_PER_BATCH]; // source masks for multi-edge transitions
    T masks_dst[N][N_NFAS_PER_BATCH]; // destination masks for multi-edge transitions
    T masks_char[N_CHARS][N_NFAS_PER_BATCH]; // masks for states that match each character
};
```

Example 7. Consider the regex r = `a(bc|de|fg|)h`. The following automaton $\mathcal{A}_1$ implements r.



In this example, we have three kinds of operations: 1 one-to-many operation, 1 many-to-one operation and 1 shift operation. The one-to-many operation corresponds to the dotted lines starting from state q_0 and going to the set of states $\{q_3, q_5, q_7\}$. The many-to-one operation, in dashed line, goes from the set of states $\{q_2, q_4\}$ to the final state q_7. The remaining transitions are of distance 1

Algorithm 5: ShiftAndOps for M shifts and N multi-edge transitions on the GPU

```
1  Kernel ShiftAndOps<T, M, N>(output, batches, text):
2      ...  // compute batchId and localNfaId as in ShiftAnd
3      &BatchSAOps<T, M, N> batch ← &batches[batchId]   // pointer to batch
4      T maskInitial ← batch.masks_initial[localNfaId]   // mask of initial states
5      T maskFinal ← batch.masks_final[localNfaId]   // mask of final states
6      T states ← 0 // set of active states
7      nMatches ← 0 // number of matches
8      for c in text do // left-to-right pass over input text
9          next ← 0   // next active states
10         for $i = 0, \ldots, M - 1$ do // take M “shift” transitions
11             mask ← batch.masks_shift[i][localNfaId]
12             d ← batch.shift_dist[i][localNfaId]
13             next ← next OR ((states AND mask) ≪ d)
14         for $i = 0, \ldots, N - 1$ do // take N “multi-edge” transitions
15             src ← batch.masks_src[i][localNfaId]   // set of source states
16             dst ← batch.masks_dst[i][localNfaId]   // set of destination states
17             if (states AND src) ≠ 0 then  next ← next OR dst
18         mask ← batch.masks_char[c][localNfaId]   // get mask for character c
19         states ← (next OR maskInitial) AND mask   // execute transition
20         nMatches ← nMatches + ((states AND maskFinal) ≠ 0)
21     globalNfaId ← batchId · N_THREADS_PER_WARP + localNfaId
22     output[globalNfaId] ← nMatches
```

Table 2. Multi-edge masks of the kernel ShiftAndOps<u32, 1, 2> for the pattern r = `a(bc|de|fg|)h`.

Operation	src	dst	mask_src	mask_dst
one-to-many (dotted line)	q_0	$\{q_3, q_5, q_7\}$	00000001	10101000
many-to-one (dashed line)	$\{q_2, q_4\}$	q_7	00010100	10000000

Table 3. Execution of the kernel ShiftAndOps<u32, 1, 2> for the pattern `a(bc|de|fg|)h` and the string `abch`.

Input	`a`	`b`	`c`	`h`
next (before line 14)	00000000	00000010	00000100	00000000
next (before line 18)	00000000	10101010	00000100	10000000
mask (after line 18)	00000001	00000010	00000100	10000000
states (after line 19)	00000001	00000010	00000100	10000000
nMatches	0	0	0	1

and are executed using 1 shift operation (solid line). Note that transition $q_0 \longrightarrow q_1$ (resp., $q_6 \longrightarrow q_7$) can be encoded either in the one-to-many (resp., many-to-one) operation or in the shift operation.

Table 3 presents the execution of $\mathcal{A}_1$ by kernel ShiftAndOps<u32, 1, 2> for the input string `abch`. ShiftAndOps can execute all the transitions leading to a single state (resp., starting from a single state) in one operation. The values of the masks used by the multi-edge transitions are shown in Table 2. For the shift transition, we have *mask_shift* = 01101011 and *shift_dist* = 1 because the states q_0, q_1, q_3, q_5, q_6 are source states of transitions of distance 1. The character masks for `a`, `b`, `c` and `h` are 00000001, 00000010, 00000100 and 10000000 respectively. After consuming the character `a`, we start by performing the shift transitions of distance 1 with

$$\begin{aligned} \mathit{next} &= \mathit{next} \text{ OR } ((\mathit{states} \text{ AND } \mathit{mask}) \ll d) \\ &= \texttt{00000000} \text{ OR } ((\texttt{00000000} \text{ AND } \texttt{01101011}) \ll 1) = \texttt{00000000}. \end{aligned}$$

To perform the one-to-many operation $q_0 \longrightarrow \{q_3, q_5, q_7\}$ we calculate

$$\mathit{states} \text{ AND } \mathit{src} = \texttt{00000000} \text{ AND } \texttt{00000001} = \texttt{00000000}.$$

Similarly, to perform the many-to-one operation $\{q_2, q_4\} \longrightarrow q_7$ we calculate

$$\mathit{states} \text{ AND } \mathit{src} = \texttt{00000000} \text{ AND } \texttt{00010100} = \texttt{00000000}.$$

This means that no multi-edge operations are applied and we have that $\mathit{next} = \texttt{00000000}$ before line 18. The value of *states* after line 19 is computed with

$$\begin{aligned} \mathit{states} &= (\mathit{next} \text{ OR } \mathit{maskInitial}) \text{ AND } \mathit{mask}) \\ &= (\texttt{00000000} \text{ OR } \texttt{00000001}) \text{ AND } \texttt{00000001} = \texttt{00000001}. \end{aligned}$$

We proceed now to the second round. After consuming the character `b`, we have that

$$\mathit{next} = \texttt{00000000} \text{ OR } ((\texttt{00000001} \text{ AND } \texttt{01101011}) \ll 1) = \texttt{00000010}$$

before line 14. This corresponds to taking the shift transition from q_0 to q_1. The one-to-many operation $q_0 \longrightarrow \{q_3, q_5, q_7\}$ is enabled (because q_0 is active in *states*), so we obtain $\mathit{next} = \texttt{10101010}$ after all transitions are performed. After line 19, we have that $\mathit{states} = (\texttt{10101010} \text{ OR } \texttt{00000001}) \text{ AND } \texttt{00000010} = \texttt{00000010}$.

For the third round, we consume the character `c`, and $\mathit{next} = \texttt{00000000} \text{ OR } ((\texttt{00000010} \text{ AND } \texttt{01101011}) \ll 1) = \texttt{00000100}$ before line 14. This corresponds to taking the shift transition from q_1 to q_2. No multi-edge operation is enabled, and therefore *next* is not updated. After line 19, we have that $\mathit{states} = (\texttt{00000100} \text{ OR } \texttt{00000001}) \text{ AND } \texttt{00000100} = \texttt{00000100}$.

For the final round, we consume the character `h`, and $\mathit{next} = \texttt{00000000} \text{ OR } ((\texttt{00000100} \text{ AND } \texttt{01101011}) \ll 1) = \texttt{00000000}$ before line 14. This means that the shift operation performs no transition. For the many-to-one operation $\{q_2, q_4\} \longrightarrow q_7$, we have that $\mathit{states} \text{ AND } \mathit{src} = \texttt{00000100} \text{ AND } \texttt{00010100} = \texttt{00000100}$. So, we take the multi-edge operation and we have before line 18 that $\mathit{next} = \texttt{10000000}$. The one-to-many operation $q_0 \longrightarrow \{q_3, q_5, q_7\}$ is not performed (q_0 is not active in *states*). After line 19, we have $\mathit{states} = (\texttt{10000000} \text{ OR } \texttt{00000001}) \text{ AND } \texttt{10000000} = \texttt{10000000}$. As $\mathit{states} \text{ AND } \mathit{maskFinal} = \texttt{10000000} \neq \texttt{00000000}$, we report a match and increment *nMatches*.

We need 13 basic operations (i.e., bitwise and memory access operations) to execute r with `ShiftAndOps`<u32, 1, 2>. We would need 31 basic operations if we used `ShiftAndDist`<u32, 7>. Table 4 has more details on the number of basic operations required for executing our kernels.

Kernel datatype. Until now, we have only considered regexes of size at most 32 (# character classes), because registers on the GPU are 32 bits wide. However, if we want to support a large number of regular expressions, we need to extend this bound. Fig. 5 presents the distributions of the number of NFA states across the datasets. The medians are: 16 for Prosite, 24 for Snort, 32 for SpamAssassin, 23 for Suricata and 48 for Yara. From the median, we already know that more than half of the regexes would not be handled with the `u32` datatype for the SpamAssassin and Yara datasets. We observe that for the Prosite dataset, the majority of regexes can be simulated using Shift-And. For the other datasets, a large portion of the regexes have more than 32 states. This shows there is a need for wider datatypes. The NVCC compiler for CUDA code supports bitwise operations up to 128 bits. For each of the 4 kernel families (Shift-And, ShiftAndDist, ShiftAndGap and ShiftAndOps), we have created variants for datatype up to `u256`.

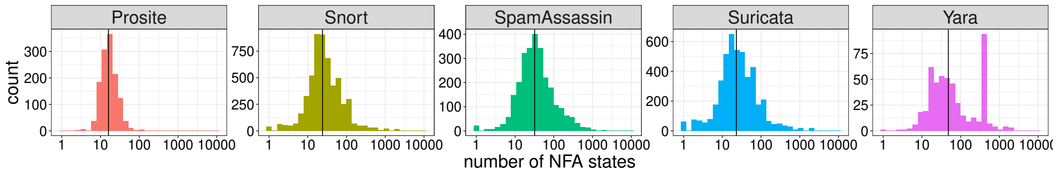


Fig. 5. Distribution of the number of NFA states per regex across the datasets. Vertical line indicates the median.

Table 4. Summary of the operations per input character to simulate the transitions for the ShiftAnd, ShiftAndGap, ShiftAndDist and ShiftAndOps kernels. We use the symbol ≪ for the left shift operation, OR for bitwise or, AND for bitwise and, and NOT for bitwise complement.

kernel	mem. access	≪	OR	AND	NOT	total
`ShiftAnd<u32>`	1	1	1	1	0	4
`ShiftAndGap<u32>`	1	1	2	4	1	9
`ShiftAndDist<u32, D>`	D + 1	D	D + 1	D + 1	0	4D + 3
`ShiftAndOps<u32, M, N>`	2(M+N)	M	M + N	M + N	0	5M + 4N

Number of operations. Table 4 summarizes the number of bitwise operations and memory accesses per input character for each kernel. We exclude the operations needed to calculate the number of matches. For `ShiftAnd` and `ShiftAndGap`, the total number of basic operations is constant. For `ShiftAndDist<D>`, it is linear in the maximum distance D. For `ShiftAndOps<M, N>`, the total number of basic operations in linear in M (# shift operations) and N (# multi-edge operations). The kernel `ShiftAnd<u32>` uses the least number of operations, namely 4. The kernels `ShiftAndGap<u32>` and `ShiftAndOps<u32, 1, 1>` both require 9 operations, thus we expect their performance to be similar. For datatypes T that are wider than `u32`, each bitwise operation requires several machine instructions. Therefore, we generally expect that doubling the width of the datatype (for instance, from 64 to 128 bits) will double the number of machine instructions executed per input character (for instance, 16 operations for `ShiftAnd<u128>`).

Kernel expressiveness. Table 5 presents a summary of the kernels' capabilities to simulate the different types of transitions mentioned earlier. We observe that `ShiftAndGap` is more expressive than `ShiftAnd` as it can execute all the transitions that `ShiftAnd` can execute in addition to the gap transitions. `ShiftAndDist` and `ShiftAndGap` are incomparable because even though `ShiftAndGap` can execute transitions with distance greater than 1, those are limited to gap transitions whereas `ShiftAndDist<u32, D>` cannot execute gap transitions with distance greater than D. When $N > 0$, the kernel `ShiftAndOps<u32, M, N>` is more expressive than `ShiftAndGap<u32>`, as a gap transitions can be expressed as a one-to-many transition. By construction, `ShiftAndOps<u32, M + 1, N>` is more expressive than `ShiftAndDist<u32, M>` as the shift transitions from 0 to M can be executed with $M + 1$ shift transitions (not necessarily for consecutive distances unlike `ShiftAndDist`) in addition to the M multi-edge transitions. Fig. 6 shows the relationships between GPU kernels using a Hasse diagram, in which kernels are ordered by expressiveness.

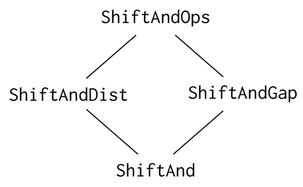


Fig. 6. Hasse diagram for GPU kernel expressiveness.

Table 5. Summary of the expressiveness of the ShiftAnd, ShiftAndGap, ShiftAndDist and ShiftAndOps kernels. ∞ means that the distance is only bounded by the number of states.

kernel	shifts	min. dist.	max. dist.	backedges	self-loops	gaps	multi-edges
`ShiftAnd`<u32>	1	1	1	✗	✗	✗	✗
`ShiftAndGap`<u32>	1	1	1	✗	✗	✓	✗
`ShiftAndDist`<u32, D>	$D + 1$	0	$+\infty$	✗	✓	✗	✗
`ShiftAndOps`<u32, M, N>	M	$-\infty$	$+\infty$	✓	✓	✓	N

4 Compilation and Optimizations

Recall that an instance of the multi-pattern matching problem consists of a set of patterns, given as regexes, and an input string over which to search for pattern occurrences. This section describes the compilation of regexes into NFA data that can be used by the GPU kernel families `ShiftAnd`, `ShiftAndDist`, `ShiftAndGap` and `ShiftAndOps` that were described in Section 3.

First, we describe the compilation process which transforms a dataset in the form of a set of regexes into masks (i.e., bit vectors) that are used for execution on the GPU. Then, we explain how we determine for each pattern whether it should execute on the CPU or on the GPU using a specific kernel. Finally, we discuss two transformations that allow us to move a regex (NFA) from a more computationally expensive kernel to a cheaper one.

Compilation procedure and execution. Initially, the input dataset is given as a set of regular expressions. HybridSA carefully chooses a list of kernels to run on the GPU. The regexes that cannot be handled by the chosen GPU kernels are executed on the CPU using Hyperscan. Since modern CPUs typically have several cores, the CPU workload is parallelized using threads. Hyperscan accepts a set of regexes as input, so no transformation of the patterns is needed. For the GPU part, kernel execution uses a representation of NFAs in terms of bit masks, as described in Section 3. For a given regular expression, the bit masks used to represent its NFA differ based on whether it is executed (for example) by `ShiftAnd`, `ShiftAndOps`<u32, 1, 1>, or `ShiftAndOps`<64, 1, 1>. The procedure that transforms a regex dataset into batches of masks is performed by a compilation library that we have implemented in Rust. It consists of 3 steps. In the first step, each regex is mapped to the cheapest kernel that can execute it. For instance, consider the regex r = `ab?c` and the kernels `ShiftAndDist`<u32, 2> and `ShiftAndDist`<u32, 3>. The compiler will choose `ShiftAndDist`<u32, 2>, as it can execute r and requires fewer operations than `ShiftAndDist`<u32, 3>. In the second step, we compute the masks for each regex and its associated kernel. For the regex r = `ab?c` and kernel `ShiftAndDist`<u32, 2>, we compute the $mask\_initial$ and $mask\_final$ masks which contain respectively the positions of initial and final states as well as $mask\_dist[d]$ for the source states of transitions of distance d between 0 and 2 and $mask\_char[c]$ containing the positions of states whose label matches the character c. In the third step, masks belonging to regexes to be executed by the same kernel variant are grouped together into batches (see Section 3) of 32 regexes to form a batch database. This database is used during GPU execution.

For the execution, our kernels are implemented in CUDA and the kernel variants are created using C++ templates. We have chosen not to generate specialized code, as it would not provide a substantial performance benefit. For a given dataset, we start by compiling the regex dataset. Then, we invoke the CUDA kernels concurrently using a Foreign Function Interface which binds the CUDA functions to Rust functions.

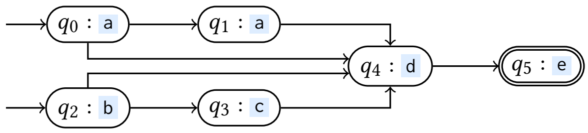
Choice of kernels to execute on the GPU. We are given a set of regular expressions. First, we need to decide which regexes to put on the GPU. Then, for regexes that are executed on the GPU, we need to choose the kernel that will execute it. For a fixed regex r, there is a list of candidate kernels. `ShiftAndOps` is the most expressive kernel, because it is capable of simulating any regex r. Nevertheless, it would not be wise to execute all the regexes on the GPU with the `ShiftAndOps` kernel. Consider, for example, $r =$ `a.{0,100}b`. Executing r with the `ShiftAndOps` or `ShiftAndDist` kernels would require many bitwise operations, resulting in poor performance. Instead, it is better to execute r with `ShiftAndGap`, which requires fewer bitwise operations in this case. In general, our compiler splits a set of regexes into two subsets: one subset that is executed on the GPU using our CUDA kernels and the other is executed on the CPU with Hyperscan. The split is done to balance the load between the CPU and the GPU. Then, for a given regex, we choose the kernel that will execute it. The strategy we use consists in selecting the kernel with the least number of operations that can simulate the regex like we did with r above. Moreover, it is inefficient to execute a kernel for only 1 NFA because the other 31 threads in the warp would not be performing useful work. If 10 regexes are compiled for `ShiftAndDist`<u32, 4> and 22 regexes for `ShiftAndDist`<u32, 5>, it is better to pack the batches by compiling all the 32 regexes for `ShiftAndDist`<u32, 5> to use a single batch instead of two. In practice, we have observed a reduction of wasted regexes (i.e. padding regexes within a batch, that do not perform any meaningful instruction) by half thanks to batch packing for the datasets considered in the experiment section.

Splitting between GPU and CPU. In our hybrid algorithm, regexes are split between GPU and CPU and executed in parallel. To attain the best performance, the durations of GPU and CPU processing should be as close as possible to prevent underutilizing the computational resources (if one finished early, then it would sit idle). We use a ***training stage*** to choose how the regexes are split between the CPU and the GPU. For a given dataset, we have collected a set of input strings that are representative of the application. The training input strings are disjoint from the input strings used for the performance evaluation. We consider different choices K_i, where K_i is a set of kernels. Every K_i splits the dataset into two parts: 1) the regexes that can be handled by K_i on the GPU, and 2) the rest of the regexes that go to the CPU. Each choice K_i has a score $\mathit{throughput}(K_i)$, which is the throughput of HybridSA for the GPU-CPU split induced by K_i. Finding an optimal CPU-GPU split is not easy, as the number of possible choices is astronomical (i.e. $2^{\#\text{ regexes}}$). We use a "fastest kernel first" strategy which gives good performance while executing in a timely manner. In this strategy, we start by examining the choice $K_1 = \{$`ShiftAnd`<u32>$\}$, then $K_2 = \{$`ShiftAnd`<u32>, `ShiftAndDist`<u32, 1>$\}$, etc. The exploration of the choices stops when we achieve peak throughput for HybridSA. This corresponds to having roughly the same computation time on the GPU and the CPU. In practice, we observe that the balance between GPU and CPU computation times is preserved when switching to the input strings of the experiments.

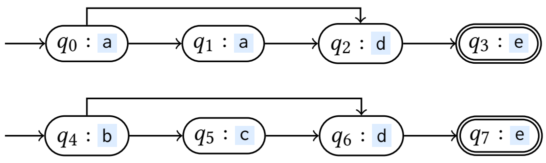
Distributivity Rewriting. We now consider the numbering problem. For a given NFA, there are several possible numbering of the automaton states, which change the operations needed to simulate it with our kernels. We perform some rewritings to reduce the number of operations to simulate an NFA and ultimately accelerate more regexes on the GPU.

Our first rewriting is at the level of the regular expression, and is based on a distributivity property that says that concatenation distributes over union. By distributivity, we can rewrite $(r_1 + r_2) \cdot r_3$ as $r_1 \cdot r_3 + r_2 \cdot r_3$. If we name m and n the number of states respectively of r_1 and r_2 , we observe that the rewriting replaced a transition of distance $m + n$ into two transitions of distance m and n. This comes at the cost of more NFA states as r_3 is duplicated. In addition to moving regexes from expensive to cheaper kernels (in terms of the number of operations), the rewriting allows for more regexes to be supported by the existing kernels, and thus to be accelerated on the GPU. In practice, we apply distributivity rewriting when it allows us to move a regex from a slower kernel to a faster kernel according to the ranking that we maintain in Table 6.

Example 8. Consider the regex r = `(a{1,2}|bc?)de`. The following automaton $\mathcal{A}_1$ implements r.



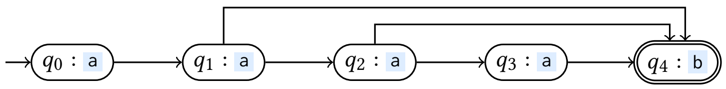
We note that the transition $q_0 \longrightarrow q_4$ has distance 4. After distributing the concatenation over the union, i.e., `(a{1,2}|bc?)de = (a{1,2}de|bc?de)`, we obtain the following automaton $\mathcal{A}_2$:



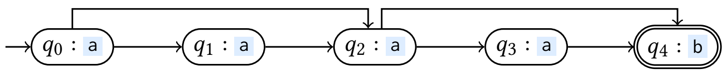
Both $\mathcal{A}_1$ and $\mathcal{A}_2$ recognize the language of r, but $\mathcal{A}_2$ reduces the maximum transition distance from 4 to 2 compared to $\mathcal{A}_1$. Thus, after the rewriting and re-labeling of the states in each of the connected components, r can be executed with kernel `ShiftAndDist<u32, 2>` instead of `ShiftAndDist<u32, 4>` before the rewriting which requires more operations (see Table 4).

Bounded repetition rewriting. In addition to the distributivity rewriting, we use another rewriting over bounded repetitions to lower the maximum transition distance. The intuition behind this rewriting is that we want to break down transitions into smaller transitions to reduce the overall distance. The rewriting is based on the following identity over regular expressions: $\sigma\{k, d \cdot k\} = (\sigma\{1, d\})\{k\}$. From a maximum distance of $(d-1) \cdot k + 1$, we go down to a maximum distance of d. This rewriting can be generalized to regexes of the form $\sigma\{m, n\}$ when $n \leq d \cdot m$ to reduce the maximum distance to d. In practice, we would choose the minimum d such that $n \leq d \cdot m$ to have the lowest possible maximum distance. With this rewriting, we are able to compile the rewritten regex with a kernel that uses fewer operations than the original regex.

Example 9. Consider the regex r = `a{2,4}b`. The following automaton $\mathcal{A}_1$ given by the Glushkov construction recognizes the language of r.



Notice that transition $q_1 \longrightarrow q_4$ has distance 3. After applying the rewriting `a{2,4}b = a{1,2}a{1,2}b` to r, we obtain the following automaton $\mathcal{A}_2$:



Both $\mathcal{A}_1$ and $\mathcal{A}_2$ recognize the language of r, but $\mathcal{A}_2$ reduces the maximum transition distance from 3 to 2 compared to $\mathcal{A}_1$.

5 Experiments

We have implemented HybridSA using the Rust programming language for the compiler and CUDA for the GPU kernels. In this section, we evaluate the performance of HybridSA to answer the following questions:

(1) What classes of regular expressions can HybridSA execute efficiently on the GPU?
(2) How well does HybridSA multi-pattern matching perform compared to existing state-of-the-art tools?

Experimental setup. The experiments were executed on a desktop machine running Ubuntu 22.04 and equipped with an Intel(R) Core(TM) i9-12900K CPU (16 cores, multithreading up to 24 threads), an NVIDIA GeForce RTX 4060 Ti (34 multiprocessors, Ada Lovelace architecture with 8 GB of global memory), and 32 GB of memory. We compile the kernels with CUDA 12.2. For each experiment, we run 10 trials and report the mean of the measurements. For the GPU measurements, we take a similar approach to previous works and focus on the execution time of the kernels, excluding memory transfers between CPU and GPU because they are negligible and can be overlapped with computations. Throughput is calculated by dividing the input size (in characters) by the execution time.

Evaluated schemes. We compare the performance of HybridSA against HotStart [Liu et al. 2020], which is a state-of-the-art engine for multi-pattern matching on the GPU, and the state-of-the-art CPU engine Hyperscan [Wang et al. 2019]. Since HybridSA uses both the CPU and the GPU, for a fair comparison we also consider a hybrid combination of Hyperscan and HotStart that we call Hyperscan+HotStart. A comparison between HybridSA and Hyperscan+HotStart, which use the same computational resources (i.e., CPU and GPU), will show the performance improvement of HybridSA due to the effectiveness of the bit-parallel kernels described in Section 3. To demonstrate the effectiveness of the optimizations presented in Section 4, we evaluate two variants of HybridSA: (1) HybridSA_no_opt, which uses the bit-parallel kernels, and (2) HybridSA_opt, which uses the rewriting and batch packing optimizations. The kernels used to execute regexes on the GPU are chosen per dataset following the compilation strategy described in Section 4.

We compare our implementation against the HotStart-MaC variant of the HotStart algorithm presented in [Liu et al. 2020], which is the version with the best overall performance. It is based on NFA execution, with various optimizations to reduce data movement and improve the compute utilization of GPU resources. Compared to HybridSA, HotStart executes all the regexes on the GPU, and the unit of computation of a thread in HotStart is the NFA state whereas, for HybridSA, a warp executes a set of 32 NFAs. Moreover, the subset of regexes that are executed by HybridSA on the GPU are input independent, meaning that the throughput is independent from the input. On the opposite, the HotStart throughput depends on the input stream. More specifically, an input stream that gives rise to many matches can degrade the overall throughput of HotStart. The HotStart algorithm performs two steps: hot state execution followed by cold state execution. Hot states are NFA states that are the most likely to get activated during matching, and in practice they are chosen to be the initial states. If many cold states are active, the second stage duration will be impacted.

To use the full computational power of the CPU, we run Hyperscan in a multithreaded setting with 24 threads, which is the number of threads maximizing the performance for the CPU we use. Hyperscan can be parallelized over the input streams or the regexes. In our settings, we parallelize over the input streams. We have observed that parallelizing over input streams has better performance compared to parallelizing over regexes. This is because the execution time of regular expressions is not uniform in Hyperscan, which can create an imbalance among the threads in the case of regex parallelism. For the combination of HotStart and Hyperscan, we select the percentage of regexes on the GPU maximizing the overall throughput, where regexes are chosen randomly.

Regex split between HotStart and Hyperscan. Finding an optimal CPU-GPU split for HotStart+Hyperscan is not easy, as the number of possible choices is enormous (i.e., $2^{\#\text{ regexes}}$). We have observed that the regexes that are expensive for Hyperscan are those that are highly nondeterministic and that match frequently. But these are also the kinds of regexes that we have noticed are expensive for HotStart. So, there does not seem to be a clear criterion for regexes that are expensive for Hyperscan but cheap for HotStart, which could be used to inform the GPU-CPU split as we did with HybridSA. Thus, we compare two choices of split: 1) split based on a training step to

Table 6. Throughput (in MB/sec) for the top 10 fastest kernels on the GPU.

kernel	throughput	datatype	d	ops
`ShiftAnd<u32>`	222	u32	0	0
`ShiftAndDist<u32, 1>`	195	u32	1	0
`ShiftAnd<u64>`	180	u64	0	0
`ShiftAndGap<u32>`	174	u32	0	0
`ShiftAndDist<u32, 2>`	160	u32	2	0
`ShiftAndDist<u32, 3>`	147	u32	3	0
`ShiftAndOps<u32, 1, 1>`	139	u32	1	1
`ShiftAndOps<u32, 1, 2>`	125	u32	1	2
`ShiftAndDist<64, 1>`	124	u64	1	0
`ShiftAndDist<u32, 4>`	124	u32	4	0

select the percentage of regexes that go to the GPU to maximize the overall throughput (labeled HotStart+Hyperscan_opt), and 2) the HybridSA's CPU-GPU split.

Benchmarks. We evaluate the performance of HybridSA over 7 applications, which contain regexes collected from real applications. These benchmarks are: (1) the **Snort** [Roesch 1999; Snort 2024] and (2) **Suricata** benchmarks [Suricata 2024] that contain patterns for network traffic, (3) the **Prosite** benchmark that includes protein motifs from the PROSITE database [Roy and Aluru 2016; Sigrist et al. 2009], (4) the **Yara** [Yara 2024] benchmarks that contain patterns that indicate the presence of viruses, (5) the **SpamAssassin** benchmark [SpamAssassin 2024] that includes patterns for detecting spam email, (6) the **Hamming** and (7) **Levenshtein** datasets containing DNA motifs extracted from the Human genome [Consortium 2022]. Compared to the AutomataZoo [Wadden et al. 2018] benchmark set, our application benchmarks are more up-to-date and contain more regexes. We use 1 GB input (split into input streams of size 8 KB) from real-world applications (PCAP packets, Spam emails, binary viruses, Human genome) for each application.

5.1 Classes of Regexes on the GPU

HybridSA excels at executing regexes with structured transition pattern that can be efficiently simulated on the GPU with bitwise operations. Fig. 7 compares the throughput performance of variants of the `ShiftAnd`, `ShiftAndGap`, `ShiftAndDist` and `ShiftAndOps` kernels on the GPU over 100 batches (i.e. 3200 regexes) and present the kernels used on the GPU per dataset for the GPU part of the hybrid algorithm HybridSA. Table 6 presents the top 10 kernels with highest throughput on the GPU over 100 batches, which corresponds to a dataset of 3200 regexes. All the kernels in the top 10 have a high throughput of more than 100 MB/s, ensuring an efficient execution on the GPU for the regexes that can be executed by those kernels. Note that our implementation is compute-bound, because the GPU simulates the execution of a large number of NFAs for each element of the input. The throughput is relative to the number of regexes in the dataset, and it would increase for a smaller dataset. Therefore, a GPU with more compute resources would enable higher throughput.

Datatype. In Table 6, which shows the top 10 kernels with the highest throughput, we only find kernels with datatypes u32 and u64. As expected, `ShiftAnd<u32>` has the highest throughput at around 220 MB/s, followed by `ShiftAndDist<u32, 1>` and `ShiftAnd<u64>`. In general, when we double the bit width of the datatype, we expect the throughput to be roughly divided by 2, given that GPU registers are 32-bit wide. However, when we go from 32-bit to 64-bit, the throughput only decreases from 222 to 180 MB/s for `ShiftAnd<u32>`, and from 195 down to 124 MB/s for

`ShiftAndDist<u32, 1>`. This indicates an optimization at the level of the hardware to support efficiently the 64-bit datatype, and suggests that our implementation can deal almost as well with NFAs of size at most 64 as with NFAs of size at most 32. For datatypes u128 and u256, the throughput is always lower than 60 MB/s. In Fig.7 (d), the compiler did not select kernels of datatype u256 and only the kernels with the highest throughput for datatype u128 (i.e., `ShiftAnd<u128>` and `ShiftAndDist<u128, 1>`). It indicates that HybridSA can execute very efficiently on the GPU NFAs that have less than 128 states and a simple transition pattern (at most distance 1) though NFAs with more states and more complex transitions can also be executed efficiently on the GPU. In fact, (d) indicates that the vast majority of regexes on the GPU are executed for datatypes u32 and u64. For Yara, all the regexes executed on the GPU are handled by `ShiftAnd<u32>` and `ShiftAndDist<u32, 1>`, two kernels of datatype u32. Those results for Yara are expected, because most of the regexes in Yara have a simple structure where most transitions are of distance 1. For Prosite, a large number of regexes (83%) are executed with `ShiftAnd<u32>`, 11% with `ShiftAnd<u64>` and the rest is executed with `ShiftAndGap<u32>`, because many regexes are of the form $ab\{0,k\}c$. For the Snort, SpamAssassin and Suricata datasets, more than 70% of regexes are handled by kernels of datatype up to u64. Those results justify our choice to consider at most the datatype u256, as the throughput for kernels with datatypes u128 and u256 is already quite low. It also indicates that the GPU may not be a great fit for very large regular expressions, which is ok because they represent only a tiny portion of the regexes in the datasets as shown in Fig. 5.

Number of operations. In Fig. 7 (b) and (c), we observe that for the 3 datatypes (u32, u64 and u128), the throughput is divided by two going from distance 1 to distance 5. In practice, the kernel with the largest distance executed on the GPU is `ShiftAndDist<u32, 9>` in Suricata. Therefore, our choice to limit the kernels to at most distance 10 is justified. Moreover, over all the datasets, the largest parameters used for ShiftAndOps are 4 shift transitions and 2 multi-edge transitions which is below the limit we have of at most 5 shift transitions and 5 multi-edge transitions. It is also consistent with (c) as the slope is flatter for kernels with 5 multi-edge operations compared to only one multi-edge operation, which indicates that executing multi-edge transitions is more expensive than executing shift transitions. The throughput is so low for the kernels `ShiftAndOps<u128, M, N>` for $M \in [0, ..., 5]$ and $N \geq 4$ that no regular expression in the datasets is executed with those kernels on the GPU. For this reason, we do not consider the datatype u256 at all for the kernel `ShiftAndOps` at compile time.

5.2 Performance Results

Fig. 8 presents the performance results achieved by the HotStart, Hyperscan, Hyperscan+HotStart, Hyperscan+HotStart_opt engines and the two variants of HybridSA: (1) HybridSA without optimizations and (2) HybridSA_opt with both the rewriting optimizations and batch packing. The horizontal axis is in log scale and is normalized to the throughput of HotStart. Table 7 presents the corresponding absolute throughputs over the datasets. Table 8 presents the compilation times for the engines over the datasets.

HybridSA against HotStart. Across all the datasets, the two variants of HybridSA outperform HotStart by at least 11× (resp., 18×) for HybridSA (resp., HybridSA_opt). The minimum speedup for HybridSA is attained for the Prosite dataset and the maximum speedup is attained for the Yara dataset with about 230×. Compared to HotStart, HybridSA only offloads regexes that can be efficiently executed on the GPU and delegates regexes with complicated access patterns to the CPU engine, Hyperscan. This approach pays off, as for all the datasets but Yara, a vast majority of regexes can still be handled on the GPU and a minority (less than 25%) is executed on the CPU. For the Hamming and Levenshtein datasets, the large number of matches makes HotStart impractical, as

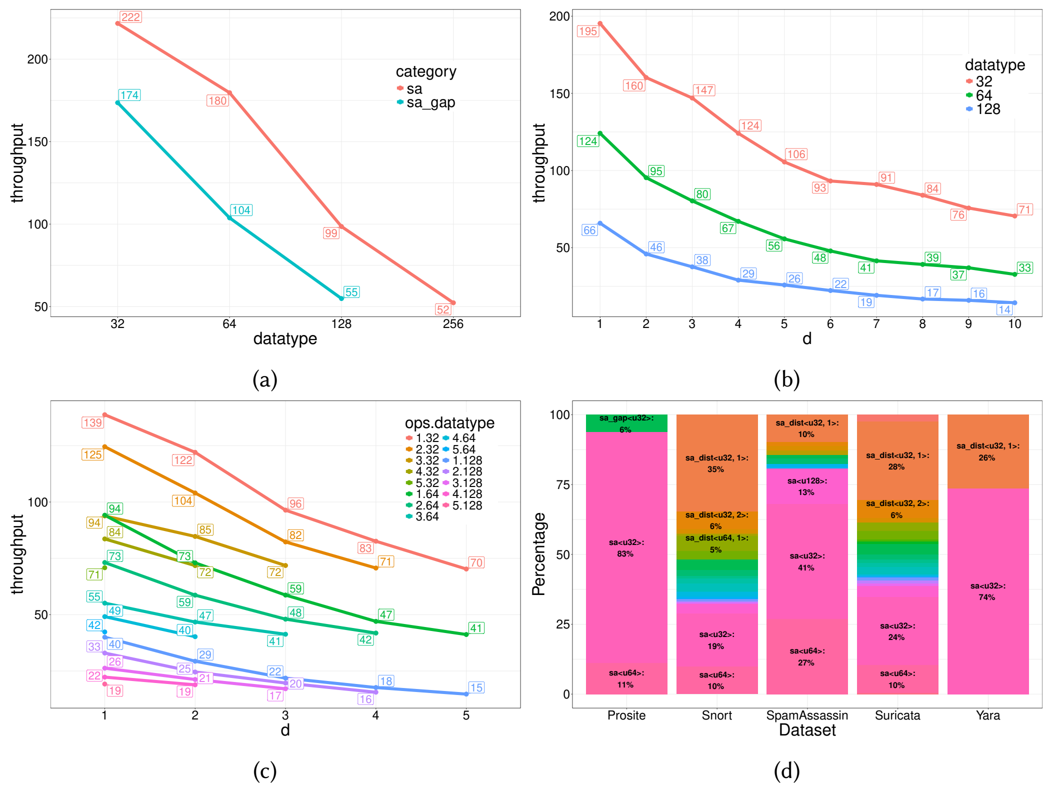


Fig. 7. Comparison of the throughput (in MB/s) for (a) ShiftAnd and ShiftAndGap, (b) ShiftAndDist for distance up to 10, (c) ShiftAndOps for up to distance 5 and 5 other operations, and (d) the kernels executed by HybridSA on the GPU. Datatypes of 32, 64, 128 and 256 bits are evaluated.

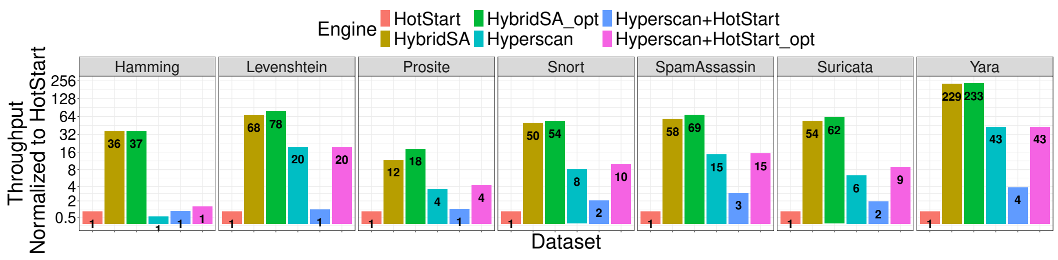


Fig. 8. Comparison of the throughput (in MB/s) with HotStart, HybridSA, HybridSA_opt, Hyperscan, Hyperscan+HotStart and Hyperscan+HotStart_opt. All bars are normalized to the throughput of HotStart.

many cold states are active at the same time. In the Yara dataset, most regexes can be supported with simple kernels such as `ShiftAnd<u32>` and `ShiftAndDist<u32, 1>` that only require a couple of bitwise operations per symbol. Thus, our bitwise algorithms are much more efficient at executing this class of regexes compared to the classical NFA simulation implemented in HotStart.

HybridSA against HotStart on the GPU. In Table 9, we compare the throughput of the optimized version HybridSA_opt and HotStart over the set of regexes that are executed on the GPU

Table 7. Absolute throughput (in MB/s) for HotStart, Hyperscan, Hyperscan+HotStart, Hyperscan+HotStart_opt HybridSA and HybridSA_opt.

engine	Hamming	Levenshtein	Prosite	Snort	SpamAssassin	Suricata	Yara
HotStart	3	4	24	2	4	2	30
Hyperscan	2	78	88	17	54	12	1300
Hyperscan+HotStart	3	5	29	4	11	4	116
Hyperscan+HotStart_opt	5	78	104	21	56	18	1300
HybridSA	117	268	290	104	216	107	6907
HybridSA_opt	119	310	450	111	253	121	7044

Table 8. Compilation time (ms) for HotStart, Hyperscan, Hyperscan+HotStart, Hyperscan+HotStart_opt and HybridSA_opt.

engine	Hamming	Levenshtein	Prosite	Snort	SpamAssassin	Suricata	Yara
HotStart	1197	558	374	58859	4203	60078	7832
Hyperscan	3140	608	1110	19826	9105	19855	2124
Hyperscan+HotStart	1236	587	670	9296	2702	8883	2023
Hyperscan+HotStart_opt	1776	624	953	18932	7702	22715	2191
HybridSA_opt	1578	359	511	9316	2537	8749	2090

Table 9. Comparison of the absolute throughput (in MB/s) between HybridSA_opt and HotStart for the portion on regexes that are executed purely on the GPU by HybridSA.

engine	Hamming	Levenshtein	Prosite	Snort	SpamAssassin	Suricata	Yara
HotStart	3	4	30	4	11	4	133
HybridSA_opt	120	314	450	110	213	113	7200

by HybridSA_opt. This way, we are able to directly compare directly the two GPU algorithms. Compared to Table 7, we observe no noticeable throughput improvement for HybridSA_opt. For HotStart, however, there is a significant performance improvement for all the datasets but Hamming and Levenshtein. This is because, for regexes with a high level of nondeterminism, the HotStart algorithm will have to process many more states, resulting in a global slowdown as more active states induce more work for the threads in HotStart. This is not a problem for HybridSA, as the performance of the GPU kernels is independent from the input text. Overall, the throughput of HybridSA_opt over the regexes it executes is still orders of magnitude better than HotStart. Those results confirm that 1) a hybrid approach is beneficial compared to pure GPU (HotStart performs better with Hyperscan) and 2) our bit-parallel algorithms provide substantial speedup on the large portion of regexes (> 75%) that it can execute efficiently.

HybridSA against Hyperscan. We observe that HybridSA_opt consistently outperforms Hyperscan across the datasets, between 4× for the Levenshtein dataset and 60× for the Hamming dataset. For the Levenshtein dataset, Hyperscan performs relatively well compared to HybridSA because the Aho-Corasick prefiltering can deal effectively with most of the regexes. However, when the number of matches becomes large, the performance of Hyperscan drops significantly as observed for the Hamming dataset where HybridSA attains the largest speedup against Hyperscan. On the contrary, HybridSA can simulate those regular expressions very efficiently, covering all the regexes with the kernel `ShiftAndDist`<u32, 9> in the case of the Hamming dataset. For the datasets that have

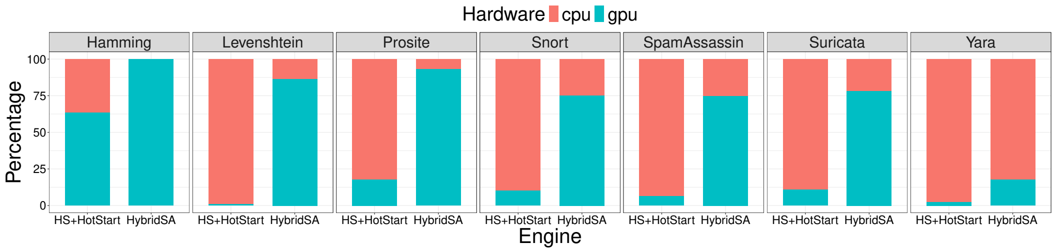


Fig. 9. Percentage of regexes that go to the GPU and CPU for the hybrid algorithms HS+HotStart and HybridSA. HS stands for Hyperscan. For HybridSA, the majority of regexes (> 75%) are executed on the GPU.

many regexes with small distances (see Fig. 4) like Snort and Yara (for the regexes executed on the GPU), Hyperscan performs relatively well compared to HybridSA because most of the regexes can be executed with `ShiftAnd<u32>` and `ShiftAndDist<u32, 1>`. For Yara and Prosite, we manage to attain a large throughput improvement from Hybrid_SA thanks to the new `ShiftAndGap` kernel because many regexes are of the form that is executed efficiently by the `ShiftAndGap` kernel. For Snort, SpamAssassin and Suricata, the throughput improvement is between 4.5× (for SpamAssassin) and 10× (for Suricata). Those datasets contain a large number of regexes and different classes of regular expressions that can be challenging for our kernels. Compared to Hyperscan which implements `ShiftAndDist`, HybridSA provides efficient execution for the classes of regexes that are supported by `ShiftAndOps` and `ShiftAndGap` kernels for datatypes up to $u256$.

Hyperscan+HotStart against Hyperscan+HotStart_opt. For all the datasets, we observe that the HybridSA GPU-CPU split performs worse compared to the *opt* variant that has a training step to select the percentage of regexes that are randomly assigned to the GPU or the CPU. The performance decrease is as high as 10× for the Yara dataset. The reason is that, for HybridSA, the GPU engine is based upon bit-parallel algorithms that are much more efficient than HotStart. When we re-use the HybridSA split in Hyperscan+HotStart, we end up with too many regexes placed on the GPU. In the end, the GPU engine runtime is much longer than its CPU counterpart, resulting in lower throughput across the datasets. By choosing a percentage of regexes to execute on the GPU, however, we are able to balance better the CPU and GPU execution time, resulting in much better throughput overall.

HybridSA against Hyperscan+HotStart_opt. We have implemented a hybrid combination of Hyperscan and HotStart to compare with our hybrid implementation HybridSA.In this version of Hyperscan+HotStart, we use a training step to choose the percentage of regexes to execute on the GPU. After the choice is made, we randomly assign regexes to the CPU or the GPU based on that percentage. For the Hamming dataset, the performance improvement of the hybrid algorithm over Hyperscan is about 2× as Hyperscan performs poorly when dealing with many matches. Compared to HybridSA, Hyperscan+HotStart is still 24× worse for the Hamming dataset. For the other datasets, the throughput of HotStart is too low in comparison to Hyperscan to provide any significant improvement. Fig. 9 presents the percentage of regexes that are offloaded to the GPU for the two hybrid algorithms, HybridSA and Hyperscan+HotStart. We observe that except for the Hamming dataset, HotStart is marginally used because its throughput is orders of magnitude smaller than Hyperscan throughput. In comparison, for all the datasets but Yara, at least 75% of regexes are handled by the GPU engine in the HybridSA algorithm. Those results also demonstrate that the HybridSA GPU kernels are able to execute efficiently more than 75% of the regexes occurring in those application datasets. In the case of Yara, the regexes are so simple and the dataset is small enough that it can be executed very efficiently on the CPU. Overall, HybridSA attains a speedup between 3.9× (for the Levenshtein dataset) and 37× (for the Hamming dataset) compared to Hyperscan+HotStart. The relatively low speedup for Levenshtein, compared to the other datasets, is due to a lower percentage of matches which favors both HotStart's approach with cold states and the Aho-Corasick prefiltering in Hyperscan. The Snort and Suricata datasets exhibit similar speedup for HybridSA_opt against Hyperscan+HotStart with a speedup of around 5×.

6 Related Work

Some regex engines provide an interface for **multi-pattern matching on CPU**, most notably Hyperscan [Wang et al. 2019], RE2 [RE2 2023] and Grep [Grep 2022]. Hyperscan [Wang et al. 2019] is the state-of-the-art multi-pattern regex matcher for modern CPUs, breaking down regular expressions into a set of strings and intricate sub-patterns to accelerate regex matching with string search. Its matching algorithm, like ours, is based on a modified version of the Shift-And algorithm accelerated with a 128-bit SIMD acceleration. Google's RE2 [RE2 2023] utilizes an on-the-fly NFA-to-DFA determinization process, caching states and seamlessly transitioning to NFA simulation when required. It maintains linear time complexity for many regex patterns, while circumventing specific constructs prone to excessive backtracking. Both RE2 and Hyperscan provide APIs for matching sets of regular expressions over an input string. Grep [Grep 2022] uses the Aho-Corasick algorithm [Aho and Corasick 1975] to perform multi-pattern matching. Besides, Saarikivi et al. (2019) [Saarikivi et al. 2019] presented the Symbolic Regex Matcher (SRM), which efficiently explores and assesses different execution paths of a regex using constraint solving and optimization techniques. Based on Brzozowski derivatives, SRM supports bounded repetition and match extraction, resembling DFA-based approaches and representing regex matching as a set of symbolic constraints. Sitaridi et al. [2016] consider SIMD acceleration on Intel CPUs.

There has been some prior works targeting **multi-pattern matching on GPU** [Avalle et al. 2016; Cascarano et al. 2010; Liu et al. 2020; Vasiliadis et al. 2011; Wang et al. 2011; Yu and Becchi 2013; Zu et al. 2012]. The first algorithm for multi-pattern matching on GPU was iNFAnt [Cascarano et al. 2010]. It uses shared memory to store the NFA active states and a symbol-first transition representation to reduce sparsity. iNFAnt employed multi-striding to enhance throughput, at the cost of higher alphabet and state counts. It was later improved on by [Yu and Becchi 2013] which proposed optimizations over iNFAnt to avoid exhaustive iteration over character class transitions and [Avalle et al. 2016] that explored further multi-striding and data compression on GPU to reduce memory accesses. Zu et al. [2012] present the notion of compatibility for NFA states. Two NFA states are compatible if they can never be active at the same time. The NFA states are thus partitioned into "compatible groups". These groups are used to simplify the access pattern of the array of active states. More recently, HotStart [Liu et al. 2020] proposed a two-step approach where the most active states, called hot states, are allocated at the beginning with their own CUDA thread whereas less active states, called cold states, are dynamically allocated threads when activated at runtime. The hot states are chosen to be the initial states, based on the assumption that very few regexes match at the same time. HotStart also introduces a compact data structure to represent NFA states alongside its transitions to fit inside the GPU registers. Meanwhile, GSpecPal [Wang et al. 2022] emerged as a speculation-centric finite state machine parallelization framework, integrating diverse speculative parallelization schemes in a latency-sensitive manner. Kargus [Jamshed et al. 2012] implements DFA execution on heterogeneous CPU/GPU hardware, where small DFAs are offloaded to the GPU. AsyncAP [Liu et al. 2023] is an extension of HotStart-MaC [Liu et al. 2020] for small datasets that match in parallel at different starting points of the input text. Even though the work is redundant, it showcases high throughput for datasets that are too small to fully occupy the GPU. Although it entails higher worst-case time complexity due to multiple reads, empirical observations on ANMLZoo and AutomataZoo databases support the practicality of this approach, as the upper bound is rarely reached in practice. [Ge et al. 2024] accelerates the NFA simulation algorithm on GPU by allowing individual threads to perform more work, effectively getting rid of the thread-level barrier at each input symbol and improving the data locality. In addition, it uses memoization to remember the initial states for each input symbol and compressed look-up tables for transitions.

When it comes to **multi-pattern matching on FPGA and ASIC**, [Sidhu and Prasanna 2001] marked the first practical application of a nondeterministic state machine on programmable logic. In later years, Yang and Prasanna [2012] put forth a novel and high-performance architecture for FPGA-based regular expression matching. Their approach involved a modular RE-NFA construction, where they first parsed regular expressions into a token list and then converted it to a modular RE-NFA suitable for FPGA implementation using a modified McNaughton-Yamada Algorithm. In this way, Yang and Prasanna were able to design two types of state update modules, one dedicated to normal character matching and the other to negated character matching, further optimizing the FPGA-based matching process. One of the most recent works is Grapefruit [Rahimi et al. 2020]. This FPGA framework for automata processing adopts compares pure BRAM and pure LUT designs, and focuses on efficient report with signal sharing. In the field of **ASIC hardware**, Kong et al. [2022] proposed an area- and energy-efficient hardware for multi-pattern matching that improved upon CAMA [Huang et al. 2022] and the AP processor [Dlugosch et al. 2014; Wang et al. 2016] to save area for bounded repetitions with an analysis called counter ambiguity. Wen et al. [2024] have recently used the model of nondeterministic bit vector automata (NBVA), introduced in [Le Glaunec et al. 2023], to handle the bounded repetition construct $r\{m, n\}$ more efficiently.

7 Conclusion

In this work, we have investigated the GPU acceleration of multi-pattern regex matching. We have proposed a heterogeneous approach that partitions the set of patterns into a subset that is matched on the CPU and a subset that is matched on the GPU. The GPU matching uses a collection of bit-parallel algorithms (`ShiftAnd`, `ShiftAndDist`, `ShiftAndGap` and `ShiftAndOps`) for the efficient simulation of NFAs. We have compared the performance of our approach against state-of-the-art CPU-based and GPU-based regex engines and observe a speedup of around 3× to 35×.

The GPU acceleration techniques of this work could possibly be used for patterns with lookaround assertions [Mamouras and Chattopadhyay 2024], as well as for match extraction [Mamouras et al. 2024] (i.e., reporting the location of a match, not just whether a match exists or not).

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. We also thank Angela W. Li for providing valuable feedback. This research was supported in part by the US National Science Foundation award CCF 2313062.

References

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (1975), 333–340. https://doi.org/10.1145/360825.360855

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

Ricardo Baeza-Yates and Gaston H. Gonnet. 1992. A New Approach to Text Searching. *Commun. ACM* 35, 10 (1992), 74–82. https://doi.org/10.1145/135239.135243

Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and

Applications. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). LNCS, Vol. 10457. Springer, Cham, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA Pattern Matching on GPGPU Devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26. https://doi.org/10.1145/1880153.1880157

Genome Reference Consortium. 2022. Genome Reference Consortium - Human Genome Overview. https://www.ncbi.nlm.nih.gov/grc/human Accessed: March 1, 2024.

Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking Large-scale Automata Processing on GPUs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 268–285. https://doi.org/10.1145/3617232.3624848

Victor Mikhaylovich Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1–53. https://doi.org/10.1070/RM1961v016n05ABEH004112

GNU Grep. 2022. GNU Grep - Global Regular Expression Print. https://www.gnu.org/software/grep/ Accessed: March 11, 2023.

Yi Huang, Zhiyu Chen, Dai Li, and Kaiyuan Yang. 2022. CAMA: Energy and Memory Efficient Automata Processing in Content-Addressable Memories. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, USA, 25–37. https://doi.org/10.1109/HPCA53966.2022.00011

Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2382196.2382232

Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. 1968. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *Commun. ACM* 11, 12 (1968), 805–813. https://doi.org/10.1145/364175.364185

Stephen Cole Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, Claude E. Shannon and John McCarthy (Eds.). Number 34 in Annals of Mathematics Studies. Princeton University Press, 3–41.

Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. ACM, New York, NY, USA, 733–748. https://doi.org/10.1145/3519939.3523456

Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching Using Bit Vector Automata. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 92 (2023), 30 pages. https://doi.org/10.1145/3586044

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 (ASPLOS '20)*. ACM, New York, NY, USA, 251–265. https://doi.org/10.1145/3373376.3378471

Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1, Article 27 (2023), 27 pages. https://doi.org/10.1145/3579453

Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 92 (2024), 31 pages. https://doi.org/10.1145/3632934

Konstantinos Mamouras, Alexis Le Glaunec, Wu Angela Li, and Agnishom Chattopadhyay. 2024. Static Analysis for Checking the Disambiguation Robustness of Regular Expressions. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 231 (2024), 25 pages. https://doi.org/10.1145/3656461

Gonzalo Navarro. 2001. NR-grep: A Fast and Flexible Pattern-Matching Tool. *Software: Practice and Experience* 31, 13 (2001), 1265–1312. https://doi.org/10.1002/spe.411

Gonzalo Navarro and Mathieu Raffinot. 2002. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press.

NVidia. 2013a. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

NVidia. 2013b. Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

NVidia. 2024. CUDA C++ Programming Guide: Programming Model. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model

Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *2020 IEEE 28th Annual International Symposium on Field-Programmable*

Custom Computing Machines (FCCM). IEEE, USA, 138–147. https://doi.org/10.1109/FCCM48280.2020.00027

RE2. 2023. RE2: Google's regular expression library. Website. https://github.com/google/re2 Accessed: March 11, 2023.

Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington) *(LISA '99)*. USENIX Association, USA, 229–238. https://www.usenix.org/legacy/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf

Indranil Roy and Srinivas Aluru. 2016. Discovering Motifs in Biological Sequences Using the Micron Automata Processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 13, 1 (2016), 99–111. https://doi.org/10.1109/TCBB.2015.2430313

Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019) (LNCS, Vol. 11427)*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer, Cham, 372–378. https://doi.org/10.1007/978-3-030-17462-0_24

Reetinder Sidhu and Viktor K. Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, USA, 227–238. https://doi.org/10.1109/FCCM.2001.22

Christian J. A. Sigrist, Lorenzo Cerutti, Edouard de Castro, Petra S. Langendijk-Genevaux, Virginie Bulliard, Amos Bairoch, and Nicolas Hulo. 2009. PROSITE, A Protein Domain Database for Functional Characterization and Annotation. *Nucleic Acids Research* 38, suppl_1 (2009), D161–D166. https://doi.org/10.1093/nar/gkp885

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 (DaMoN '16)*. ACM, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2933349.2933357

Snort. 2024. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/ Accessed: March 11, 2024.

Apache SpamAssassin. 2024. Apache SpamAssassin. https://spamassassin.apache.org/ Accessed: March 11, 2024.

Suricata. 2024. Suricata - Open Source Intrusion Detection and Prevention Engine. https://suricata.io/ Accessed: March 11, 2024.

The PCRE2 Developers. 2024. Perl-compatible Regular Expressions (revised API: PCRE2). https://pcre2project.github.io/pcre2/doc/html/index.html.

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*. IEEE Computer Society, USA, 216–225. https://doi.org/10.1109/IISWC.2011.6114181

Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, New York, NY, USA, 13–24. https://doi.org/10.1109/IISWC.2018.8573482

Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An Overview of Micron's Automata Processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES '16)*. ACM, New York, NY, USA, Article 14, 3 pages. https://doi.org/10.1145/2968456.2976763

Lei Wang, Shuhui Chen, Yong Tang, and Jinshu Su. 2011. Gregex: GPU Based High Speed Regular Expression Matching Engine. In *Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS '11)*. IEEE Computer Society, USA, 366–370. https://doi.org/10.1109/IMIS.2011.107

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 *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

Yuguang Wang, Robbie Watling, Junqiao Qiu, and Zhenlin Wang. 2022. GSpecPal: Speculation-Centric Finite State Machine Parallelization on GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, USA, 481–491. https://doi.org/10.1109/IPDPS53621.2022.00053

Ziyuan Wen, Lingkun Kong, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2024. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/3620665.3640412

Sun Wu and Udi Manber. 1992. Fast Text Searching: Allowing Errors. *Commun. ACM* 35, 10 (oct 1992), 83–91. https://doi.org/10.1145/135239.135244

Yi-Hua Yang and Viktor K. 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

Yara. 2024. ClamAV - The pattern matching swiss knife for malware researchers. Website. https://virustotal.github.io/yara/ Accessed: August 10, 2024.

Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS '06)*. ACM, New York, NY, USA, 93–102. https://doi.org/10.1145/1185347.1185360

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 (CF '13)*. ACM, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/2482767.2482791

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. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/2145816.2145833

Received 2024-04-06; accepted 2024-08-18