Accelerating Regular-Expression Matching on FPGAs with High-Level Synthesis

by

Devon Callanan

B.S. Computer Engineering, University of Pittsburgh, 2019

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical and Computer Engineering

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Devon Callanan

It was defended on

April 2, 2021

and approved by

Amr Mahmoud, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Samuel Dickerson, Ph.D., Director and Assistant Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Alan George, Ph.D., Department Chair and Professor, Department of Electrical and Computer Engineering

Accelerating Regular-Expression Matching on FPGAs with High-Level Synthesis

Devon Callanan, M.S.

University of Pittsburgh, 2021

The importance of security infrastructures for high-throughput networks has rapidly grown as a result of expanding internet traffic and increasingly high-bandwidth connections. Intrusion-detection systems (IDSs), such as SNORT, rely upon rule sets designed to alert system administrators of malicious packets. Methods for deep-packet inspection, which often depend upon regular-expression searches, can be accelerated on programmable-logic (PL) architectures using non-deterministic finite automata (NFAs). Prior designs have relied upon register-transfer level (RTL) design descriptions and have achieved efficient resource utilization through fine-grained optimizations. New advances made by field-programmable gate array (FPGA) vendors have led to powerful compiler toolchains for OpenCL and SYCL that allow for rapid development on PL architectures while generating competitive designs in terms of performance. The goal of this work is to evaluate performance differences between a custom, SYCL- and OpenCL-based, acceleration architecture for regular expressions and comparable RTL-based designs. The simplicity of the application, which requires only basic hardware building blocks, adds to the novelty of the comparison. In contrast to prior RTL-based solutions, which show frequency degradation with bandwidth scaling, this approach is able to maintain stable and high operating frequencies at the cost of resource usage. By scaling input bandwidth with multi-character transformations, high-throughput designs can be realized. Using Intel's OpenCL compiler, throughputs in excess of 17 Gbps can be achieved on Intel's Arria 10 Programmable Acceleration Card and 19.4 Gbps with Intel's Stratix 10 Programmable Acceleration Card, outperforming similar designs with RTL, as reported in the literature. SYCL-based designs, synthesized with Intel's oneAPI compiler show performance degradation but still achieve higher throughput, up to 15.6 Gbps, than

past RTL-based implementations. Overall, OpenCL and SYCL development yields both competitive results, when compared to the fine-grained RTL development process, and many ease-of-use improvements and design abstractions.

Table of Contents

List of Tables

List of Figures

Preface

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

The author would like to thank his coworkers at NSF-SHREC Pittsburgh, especially Luke, Kyle, Michael, Alex, and Colin for their tireless work in supporting this scholarship. Additionally the author would like to thank his friends and family for their constant support.

1.0 Introduction

Enterprises around the world lose billions of dollars annually from the direct and indirect costs of cyber attacks against intellectual property, service availability, and user trust [1]. Security measures designed to defend against cyber threats must support higher-throughput networks, catch more nuanced attacks, and present a more cost-effective model as global connectivity grows. One popular approach to enterprise security is a network intrusion detection System (NIDS) such as SNORT [2]. SNORT inspects each incoming packet for malicious content and flags it accordingly. To maintain high quality-of-service (QoS), the NIDS must parse packets in real time. Failure to match the throughput of the link leads to network degradation, dropped packets, or unsearched packets in the trusted network. The advent of enterprise-level 100G Ethernet connections makes real-time packet inspection more critical and matching network speeds more challenging.

An important stage in an NIDS pipeline is the application of regular-expression (regex) matching on packet payloads. The rules, based on Perl Compatible Regular Expressions (PCRE), separate potentially malicious traffic from benign traffic through deep packet inspection. After pre-processing and stream reassembly, a series of content filters, including the regex rules, are applied to the packets. CPU-based implementations often rely on recursive algorithms which must backtrack on a failed match, exhaustively trying each combination for concurrent quantifiers. This can become a bottleneck in the networking infrastructure. Some research has shown that specially formed packets can target recursion-based NIDS and take up to 1.5 million times longer to inspect than benign packets [3]. Such a denial-of-service (DoS) attack could completely disable NIDS security layers with ease.

To combat the disadvantages of traditional regular-expression matching approaches, deterministic and non-deterministic finite automata (DFAs and NFAs) are often used as fast and reliable alternatives. Regexes define regular languages and, for any expression, a finite automaton can be constructed to accept only that regular language defined by the expression. Automata perform searches with time complexity independent of input text content by concurrently investigating paths and maintaining state information, eliminating the need

to backtrack across match possibilities. These algorithms can be efficiently realized on various accelerator platforms: GPUS [4], FPGAs [5], and custom automata hardware, such as Micron's Automata Processor [6].

Both DFAs and NFAs suffer from limitations caused by their poor scalability. DFAs show state explosion which affects their ability to store large rule sets [7, 8] and the space complexity of NFAs grows as they are scaled for higher bandwidths. NFAs can be transformed, each time doubling the number of input characters which they consume during a transition. Such a transformation increases their complexity, doubling the number of possible paths through the automata. This growth places an upper bound on their use in large packet inspection systems. To alleviate this pressure and allow for further scaling of throughput to meet the demands of enterprise networks, packet pre-filtering techniques, which leverages the sparsity of intrusion attempts, have been investigated.

FPGAs in particular have been explored extensively as an avenue for fast regex-matching acceleration [7, 9, 10, 5, 11, 12]. Their reconfigurability allows for updates to NIDS rule sets as new threats are identified, cutting costs compared to application-specific integrated circuits (ASICs). FPGAs are also well suited for streaming apps, such as network security, due to their ability to synthesize custom, and often low-latency, data pipelines that can process data in place, rather than relying on memory for storage of the working set.

FPGA development is often done on the RTL level, which requires intimate knowledge of hardware design paradigms. Device manufacturers often offer an alternative design flow based on high-level synthesis (HLS). HLS allows developers to write applications in high-abstraction languages, such as OpenCL or SYCL. These tools help in rapid prototyping and improve ease-of-use at the cost of access to fine grained optimizations.

This work aims to implement regexes as NFAs on FPGAs using HLS. Compiler toolchains for OpenCL and SYCL were used to synthesize a representative expression set from a popular NIDS. Implemented on both Intel's Arria 10 and Stratix 10 Programmable Acceleration Cards (PACs), we achieve maximum throughputs of 17.88 and 19.40 Gbps respectively for the OpenCL-based designs and a throughput of 15.6 Gbps using SYCL on the Arria 10 PAC. The following lists the key contributions of this work:

- A comprehensive regex acceleration engine capable of handling a wide range of expression syntax
- A holistic investigation of real-world regex acceleration and feasibility of scaling
- An implementation of NFAs for regex searching realized completely in OpenCL and compiled for FPGA
- A comparison of HLS compilers and target platforms

2.0 Background

This section provides general information in three parts. First, the target app and the use of regexes in network security is reviewed. Then, finite automata and their relationship with regexes is explored. Lastly, the design process on FPGAs is investigated, touching on RTL- and HLS-based approaches.

2.1 SNORT

Regexes are pattern descriptors that allow for concise articulation of complex search queries. Libraries such as the popular PCRE [13] expand the simple mathematical confines of regexes to include convenient functionality for app developers. SNORT uses the PCRE2 standard as the basic syntax for regular-expression matching in its rules.

To accomplish the goal of "real-time traffic analysis and packet logging," [2] SNORT coalesces payloads across multiple packets to form a complete request, then compares that request against sets of rules designed to classify network traffic. A rule contains information on the action to be triggered if a match occurs and options such as *content* or *pcre* which narrow down the set of packets that can trigger the action. The *pcre* option allows rule creators to specify a regex as match criteria. Severe threats can trigger notifications while less pressing events can be logged for later review.

2.2 Non-Deterministic Finite Automata

Finite automata are commonly used for the acceleration of regexes due to their advantage in computational complexity over comparable recursive approaches. Additionally, backtracking is difficult to realize on programmable-logic (PL) due to memory dependencies that significantly affect parallelism and lengthen critical paths. As state machines, finite au-

tomata largely sidestep these challenges due to their memory-less nature. Mathematically, finite automata consist of a set of states and a transfer function F, as seen in (2.1), that accepts an input I and the current active states q to produce a set of the next active states [9]. Once an input is consumed, no historical references to it need be maintained.

$$q_{i+1} = F(I, q_i) \tag{2.1}$$

Deterministic finite automata (DFAs) only allow a single next state to result from the transfer function while NFAs allow for a set of next states. DFAs and NFAs can represent the same regular languages; however, DFAs need exponentially more states than an equivalent NFA [7, 8]. This state explosion makes them unfavorable on FPGAs where each state requires additional resources.

Regular expressions and NFAs can be designed to accept the same regular language. The process of converting a regex into an NFA is detailed by Thompson [14] and McNaughton-Yamada [15]. Thompson's construction, used in this work, results in an NFA with both transitions depending on a criterion, and ϵ-transitions that activate with no input criteria. Each NFA can be displayed as a set of states connected by one way transitions, each governed by a criterion or labeled as an ϵ-transition as seen in Fig. 1.

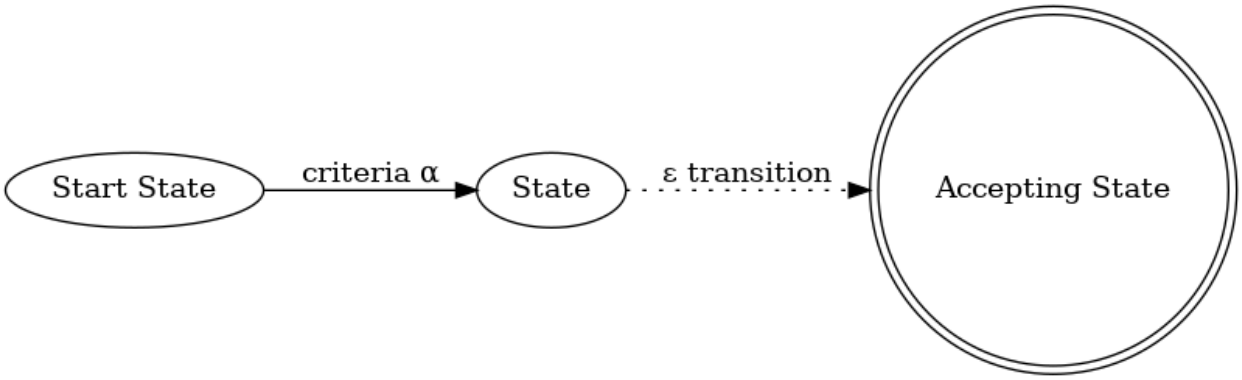

Figure 1: Anatomy of a non-deterministic finite automaton

2.3 FPGA Development

FPGA-based apps are traditionally developed on the register transfer level (RTL) with hardware description languages such as Verilog or VHDL. HLS tools have improved, making the use of abstract functional descriptions in the design of PL viable. Apps are developed in high-abstraction languages, which are more familiar to software developers, and then passed through intelligent compilers which abstract the hardware description process. Device vendors develop compilers specific to their own FPGA products. These methods lower design complexity and shorten the development cycle at the cost of access to fine-grained optimizations.

One such high-level abstraction language, OpenCL, is a popular standard for heterogeneous and parallel computing used as the preferred high-level description language on Intel's FPGA platforms. The standard provides convenient constructs for accelerator memory management, kernel launching, and parallelization [16]. Intel's FPGA SDK for OpenCL has complete support for OpenCL version 1.0 and preliminary support for OpenCL version 2.0 at the time of publication [17, 18].

With the growing adoption of HLS and heterogeneous computing, Intel is building on top of their existing OpenCL tools to provide a single-source accelerator programming standard based on SYCL [19], called Data Parallel C++ (DPC++) which supports modern ISO C++. They provide a compliant DPC++ compiler along with specialized libraries under the oneAPI name. SYCL leverages accelerator APIs such as OpenCL to translate single source C++ to accelerator-specific binaries [19].

3.0 Related Work

The genesis of NFA-based regular-expression searching comes from the work of Thompson [14] and McNaughton-Yamada [15]. These investigations introduced construction algorithms used to convert a regex into an NFA. Since the 1960s, when these breakthroughs were introduced, the need for high-throughput pattern matching has only increased.

In the early 2000s, PL was used to implement regular-expression acceleration by Sidhu [10], whose tool generated a custom hardware description from each regex. This work was then extended by Yang [5] to include a number of optimizations improving both the resource requirements of the design and the overall throughput. Through the use of shift registers to mitigate character repetition and a strategy for spatially stacked multi-character matching, the design achieved 10 Gbps throughput at a cost of only a single lookup table (LUT) per state. Another approach used to achieve high throughput is presented by Yamagaki [11]. In this work, NFA transitions are transformed in a process that enables the matching of multiple input characters in a single operation. These designs reached 8 Gbps throughput.

Other works have followed a similar strategy of synthesizing generated RTL for each regex while focusing on optimizations to lower the overall resource usage to accommodate more expressions on a single device. Long [12] implements a custom hardware block for better handling of constrained repetitions and achieves up to 1 Gbps throughput. Sourdis [20] leverages prefix sharing among common expressions, realizing the redundant portion of any NFAs only once on the FPGA, to lower resource utilization.

Additional approaches have focused on alternative hardware platforms or PL designs that allow for dynamic loading of regexes. Cascarano [4] and then Zu [21] have investigated the use of GPUs to varied levels of success, achieving up to 13 Gbps on small-scale expression sets. The Micron Automaton Processor has been used to accelerate pattern matching [6] at throughputs of 1 Gbps. A novel FPGA approach focused on throughput enabled small NFAs of only tens of states to be dynamically loaded into device memory and processed at 40 Gbps [9] by employing a clever pipelining approach through the use of parallel prefix evaluation.

Still other works attack deep packet inspection through a filtering pipeline which isolates packets with a high probability of a match before applying a regex verifier. Some designs focus on plaintext signatures which allow for the use of hash tables for match verification [22, 23]. Taking a similar pipeline approach but targeting regex, Bando [24] tackles the issues of effectively filtering regex signatures with low specificity but does not detail a solution for verifying filtered packets once identified.

4.0 Approach

The proposed regex accelerator design has two parts, NFA construction and OpenCL-based kernel generation. App efficiency comes from optimizations in both domains. This work draws extensively on well-proven algorithms for finite automaton generation and modification while providing an innovative OpenCL-based FPGA kernel.

Table 1: Supported regular expression operators

Op	**Name**	**Ex.**	**Meaning**
NA	Concatenation	ab	*a then b*
*	Kleene Closure	a*	*zero or more of a*
+	One Or More	a+	*one or more of a*
?	Optionality	a?	*zero or one a*
—	Union	a—b	*a or b*
()	Grouping	(ab)c	*a, b then c*
.	Dot	.	*any input*
{m}	Constrained Repetition	a{5}	*five a's*
{m,}	Minimum Repetition	a{3,}	*three or more a's*
{m,n}	Range Repetition	a{2,6}	*between two and six a's*
[...]	Character Class	[abc]	*a, b or c*
[^...]	Inverted Character Class	[^abc]	*not a, b or c*
\	Escape Character	*	*literal star character*

4.1 NFA Construction

To convert any arbitrary regex from PCRE format to a set of states and transitions in an NFA, the expression must be parsed into tokens and operators. Following the PCRE2 spec-

ification, escape characters are applied and the supported operators (Table 1) are tokenized along with the remaining non-meta characters. The parsed expression is then re-ordered into postfix notation using the shunting-yard algorithm tuned with PCRE operator precedence [25].

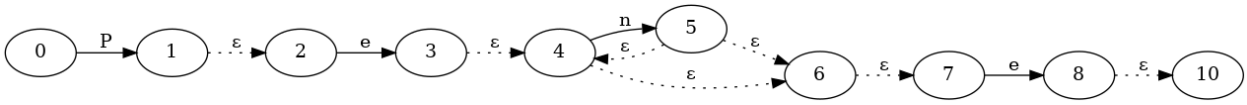

Figure 2: State machine from Thompson's construction for "Pen*e"

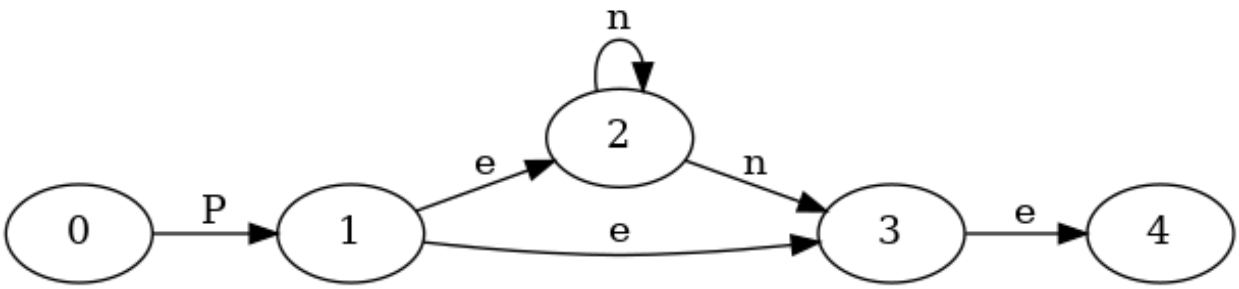

Figure 3: Minimized state machine for "Pen*e"

The Thompson construction algorithm then uses a push-down stack to systematically build the corresponding NFA from basic blocks, connecting states to one another. Fig. 2 shows NFAs resulting from the Thompson construction which contain transitions that activate with no input, called ϵ-transitions. A recursive method shown in Algorithm 1 minimizes the NFA by performing a depth-first search across all nodes and extending criteria-bound transitions to all nodes reachable by ϵ-transitions. The resulting automaton, seen in Fig. 3, no longer contains ϵ-transitions and has a reduced set of states. Removing these ϵ-transitions ensures that each transition consumes an input. This is an important characteristic for designing the accelerator as it brings uniformity to the accelerator logic and reduces the variability of the execution path.

The final step in NFA construction is applying an NFA transformation that allows for increased input bandwidth. As shown in Yamagaki [11] and Becci [26], arbitrary NFAs can

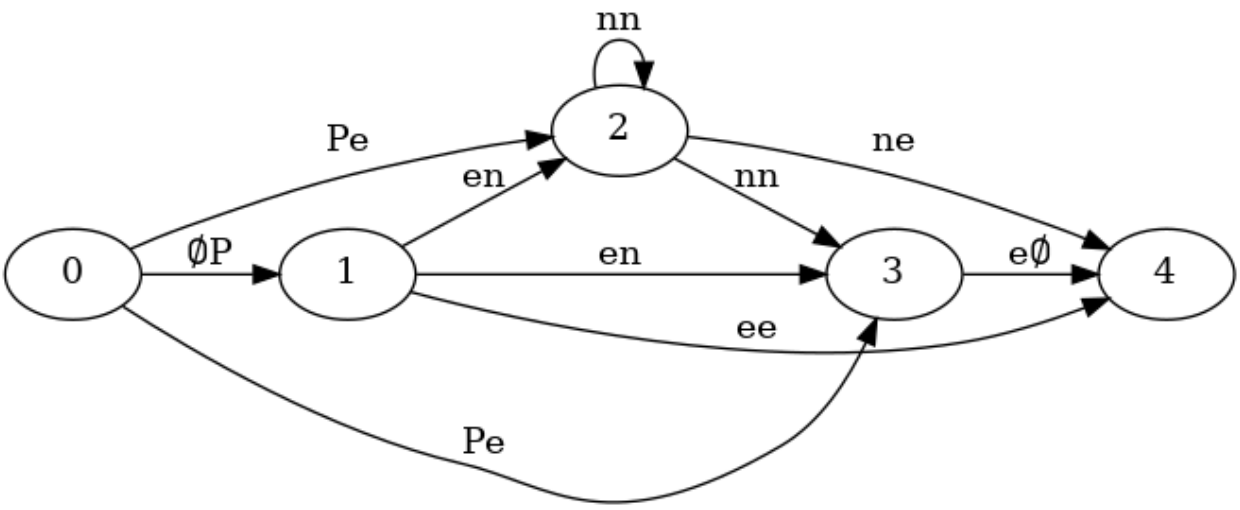

Figure 4: Two character state machine for "Pen*e"

be modified in place to consume twice the input criteria on a single transition (Fig. 4). The transformation, seen in Algorithm 2, is adopted from [11] with only minor modifications. A $\emptyset$ criteria is added for alignment purposes where the matching text pattern may start or end at an offset. Applying this transformation N times, an NFA can be made to accept 2^N input characters per transition with no additional states.

4.2 OpenCL Kernel Generation

The second phase builds an OpenCL-based kernel for the accelerator from NFAs created during the automaton construction and optimization process. Python and the templating engine Jinja2 [27] are used for the task of source generation. Automatic source generation is a feature of many prior regex accelerators. However, this work is unique in that the generated source is in OpenCL and not VHDL or Verilog. HLS has remained virtually untested in this app domain despite its proven effectiveness in complex, high-throughput apps [28].

The autogenerated kernels follow a similar architectural approach as the work of Yang [5] or Yamagaki [11] and treat each NFA as a set of states (*memory*) and transitions (*combinational logic*). While RTL designs utilize building blocks containing flip-flops and logic gates, the high-level constructs available in OpenCL allow for the use of arrays of boolean values for NFA states and *if/else* statements providing the functional description of the state transitions.

4.2.1 Memory

The OpenCL compiler creates an efficient memory unit that loads packet data from global memory, abstracting the complicated nuances of FPGA memory interconnects. The burst-coalesced memory unit outperformed other implementations by allowing a higher maximum frequency in the design. The only remaining memory structures are the active-state arrays for *current* and *next* active-state sets and the output array for reporting matches. These can be described simply as boolean vectors using standard data types in OpenCL and implemented as local variables to leverage fast on-board registers or memory logic array blocks (MLABs).

4.2.2 Combinational Logic

What remains of the kernel describes the combinational logic and interconnection of states that make up the transitions of an NFA. Early designs relied on comparisons of the input placed directly in the *if* expression. Current design iterations separate the comparison logic from the state activation logic to ease development and to de-duplicate comparators. An NFA with repeated characters, such as "Cavatappi" should only need a single comparator for the letter 'a'. It can be reused across all transitions with the same criteria. Interestingly, this optimization lead to an insignificant change in resource utilization, suggesting that the Intel OpenCL compiler already reuses redundant hardware. Despite the lack of a performance advantage, the separation was maintained as it allowed for easier development of character class comparators. Each custom character class found in an NFA is given an ID

and implemented as a boolean variable that can be evaluated on each new input. The kernel in Algorithm 3 shows both a character-class (*char_class1*) and a single-character comparator (*cmp_z*) in the current partitioned design scheme.

The boolean results of the comparators are then used in an *if* expression to activate the correct set of next states. When a multi-character NFA is used, each comparator is duplicated k times where k is the width of the multi-matching, allowing for concurrent matching of the entire input bandwidth. Finally, the vector containing the next states is copied into the vector for the current states and reinitialized in preparation for the next input character.

Algorithm 1: Minimize NFA through ϵ-transition removal

Input : NFA with ϵ-transitions

Output: NFA with a criteria on each transition

```
1  visited [];
2  foreach node i in NFA do
3      foreach node j having edge from i do
4          bypass_epsilon(i, j, crit(i,j));
5      end foreach
6  end foreach
7  remove edges with ϵ
8  Procedure bypass_epsilon(node base, node next, criteria base_crit)
9      if next is in visited then
10         add transition from base to next with criteria base_crit;
11         return;
12     end if
13     add next to visited;
14     foreach node k having edge from next do
15         if crit(next,k) is ϵ then
16             bypass_epsilon(base, k, base_crit)
17         else
18             add edge from base to next with criteria base_crit;
19         end if
20     end foreach
21     remove next from visited;
22     return;
```

Algorithm 2: Multi-character transformation presented by Yamagaki [11]

Input : NFA processing n characters per transition

Output: NFA processing $2n$ characters per transition

```
1  add n self edges labeled ∅ to initial node;
2  add self edge labeled ∅ to the final node;
3  foreach node k in NFA do
4      foreach node i having edge to node k do
5          foreach node j having edge from node k do
6              add new edge from i to j;
7              concatenate criteria of edges (i,k) and (n,j);
8              add above criteria to edge (i,j);
9          end foreach
10     end foreach
11 end foreach
12 remove original graph edges and edges from lines 1 and 2
```

Algorithm 3: OpenCL-Based NFA Kernel Accepting $[Or]zo$

Input : The data stream of characters, *stream*

Match: Indication of a match found, *match*

```
1  copy(src, dest);
2  clear(states);
3  /* State storage                                                    */
4  curr [4];
5  next [4];
6  match;
7  foreach c in stream do
8      /* Comparators                                                  */
9      bool char_class1 = c == 'O' —— c == 'r';
10     bool cmp_z = c == 'z';
11     bool cmp_o = c == 'o';
12     /* Transitions                                                  */
13     if char_class1 is true then
14         next[1] = active;
15     end if
16     if curr[1] is active and cmp_z is true then
17         next[2] = active;
18     end if
19     if curr[2] is active and cmp_o is true then
20         next[3] = active;
21     end if
22     if curr[3] is active then
23         match = found;
24     end if
25     copy(next, curr);
26     clear(next);
27 end foreach
```

5.0 Evaluation

Effective evaluation must take a holistic approach to the target app. Real-world data is especially important for performance comparisons, and in the case of NIDS, this data can vary significantly from rule set to rule set. The SNORT IDS was chosen for its size, popularity, and active development. Security researchers and industry professionals frequently update SNORT rule sets as new vulnerabilities emerge. At the time of publication, the SNORT version 2.9.15 rule set contains over 9,000 rules containing a PCRE search query, up from over 2,500 in 2012 [5]. These rules are broken down into subsets that target specific types of security vulnerabilities such as browser, application, or file-based attacks.

One notable challenge that real-world data presents is its penchant for non-conformity and unpredictability. Researchers have struggled to efficiently convert expressions containing backreferences and capture groups to NFAs and often implement special workarounds for long character repetitions. This work excludes backreferences, capture groups, and anchors and takes a naive approach to repetitions. Even with these limitations, 81% of all SNORT rules containing regexes can be converted using the tools developed in this work. As a representative set, the *file-office* SNORT sub rule set was chosen for all data collection. Details about this rule set can be found in Table 2.

The resulting NFAs were compiled for the Arria 10 PAC (A10PAC) using Intel's OpenCL SDK for FPGA version 19.4 with Quartus version 19.2 together with the Intel Acceleration Stack (IAS) version 1.2.1, and the oneAPI compiler with SYCL version 20.3. Additionally, the Stratix 10 PAC (S10PAC) was targeted with the OpenCL SDK for FPGA version 19.2 along with the IAS 2.0.1. The acceleration stacks and compilers used represent the most up-to-date software stacks supported on the respective boards. Each stack is unique to the PAC it is supported on and, despite numbering conventions, IAS 1.2.1 for the Arria 10 PAC contains a more up to date OpenCL SDK for FPGA. These devices served as the target platforms for all tests. Four versions of the accelerator were generated and compiled, covering

single-width, double-width, quad-width, and octa-width multi-character NFAs. The input bandwidth of the kernel relates directly to the width of multi-matching, from one byte in the single width and eight bytes in the octuple width.

Due to the lack of onboard networking interfaces for PACs, which would allow the FPGA to act as an inline packet inspector, this work only explores the potential for acceleration of a CPU-based IDS. All test data is copied to global memory on the accelerator prior to collecting timing results. Specially crafted network packets were used for verification and 11MB of generic text from Project Gutenberg [29] was used for throughput measurements.

Table 2: File-office metrics

File-Office Rule Set Key Metrics	
Description	This category contains rules for vulnerabilities present inside of files belonging to the Microsoft Office suite of software. (Excel, PowerPoint, Word, Visio, Access, Outlook, etc) [2]
Rules	113
Supported by this tool	95
NFA States	14709

6.0 Results

The following section presents the performance characteristics of the OpenCL-based app described above. Design throughput and maximum kernel frequency are showcased, followed by FPGA resource usage. A discussion of the implications of these results follows in the next section.

6.1 Maximum Design Frequency and Throughput

Intel's OpenCL and oneAPI compilers chooses a maximum operating frequency based on the design's critical path. In the case of the OpenCL-based designs generated for the Arria PAC, a consistent maximum frequency is achieved across all four kernel variations for each hardware target. The larger designs, despite wider bandwidths, match the maximum fre-

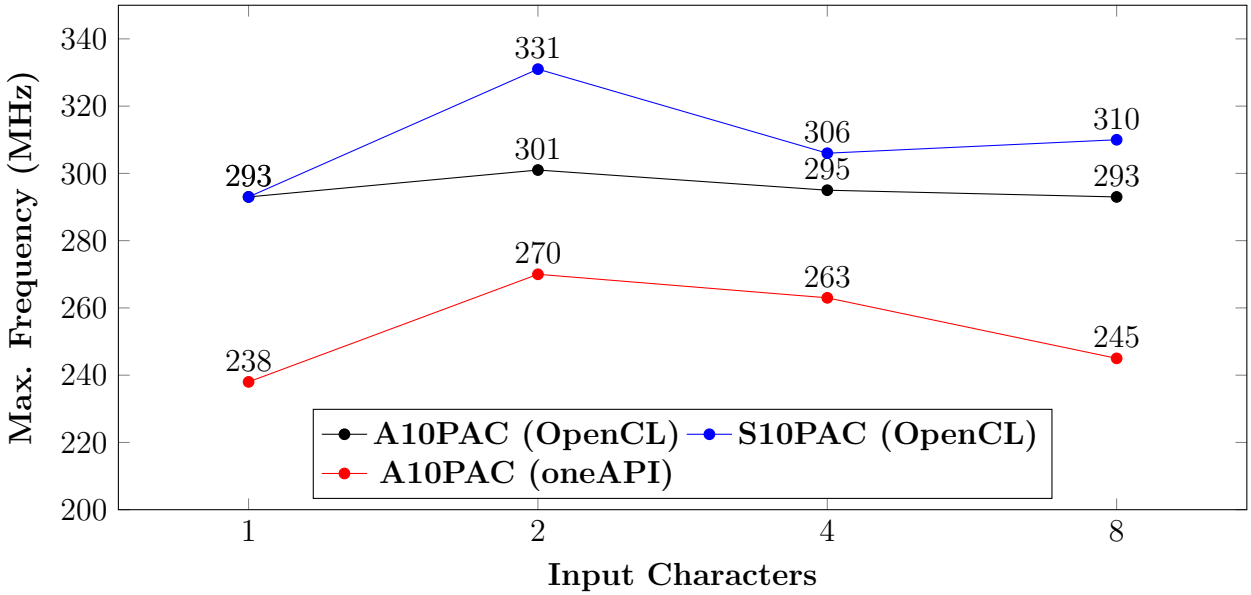

Figure 5: Maximum kernel frequency compared across input-bandwidths

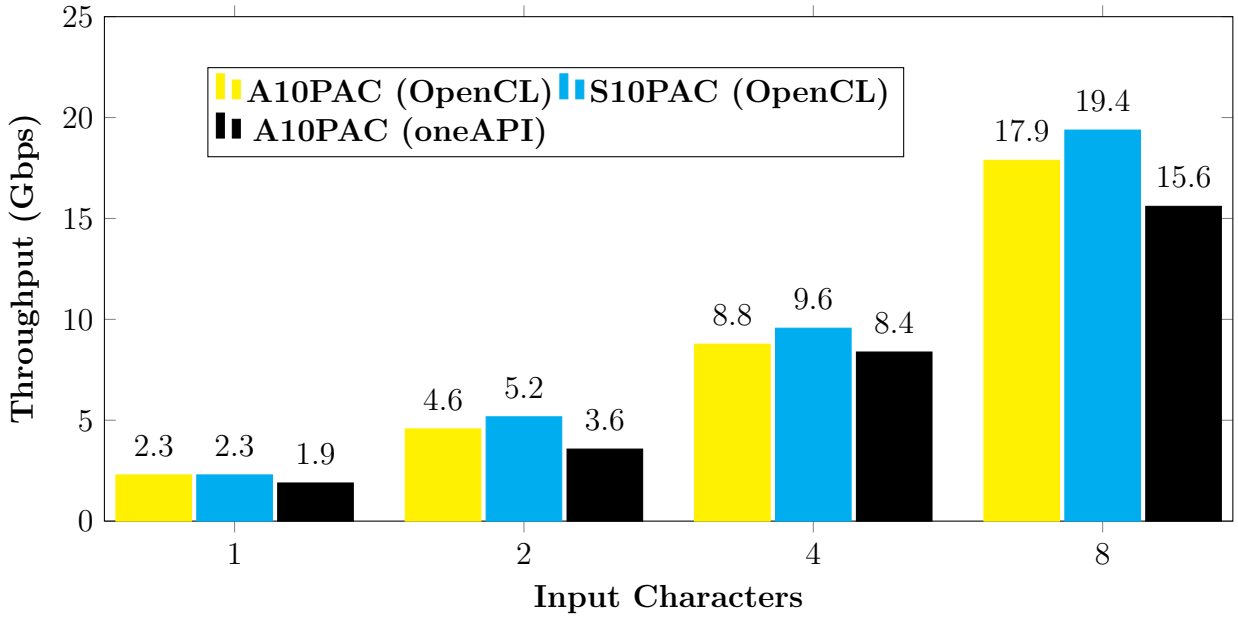

Figure 6: Results of bandwidth scaling on design throughput

quency of the single byte kernel, around 300 MHz (Fig. 5). This stability allows for nearly linear scaling of throughput with input bandwidth. As seen in Fig. 6, a 4× increase in bandwidth from two characters to eight leads to an increase in throughput by 3.9×. A maximum throughput of 17.88 Gbps was reached by the eight-character matching kernel. The Stratix PAC shows greater fluctuation of maximum operating frequency but attains higher clocks than the Arria PAC in most cases. This, in turn, leads to a maximum throughput of 19.4 Gbps. The oneAPI-based designs show results that under-perform compared those of both OpenCL software stacks, showing both greater variability in maximum frequency and lower throughput designs, reaching only 15.60 Gbps.

6.2 FPGA Resource Usage

FPGA designs must be evaluated by their resource usage along with their throughput to fully understand the efficiency and scalability of the architecture. Fig. 7 shows the total ALUT usage of the kernel at each input bandwidth. The OpenCL-based designs for the Arria 10 PAC and Stratix 10 PAC perform similarly, with the former slightly edging out the latter. Despite also targeting the the Arria 10 PAC, designs created with the oneAPI compiler use many more LUTs, at times nearly 2× as many as the OpenCL-based designs on the same device. Some of this difference can be attributed to the inference of shift registers by the OpenCL compiler. In prior work [5], shift registers are manually inserted for character repetition in an attempt to reduce the LUTs needed for these constructs. In this work, similar optimizations are implemented automatically by Intel's OpenCL compiler for repetitions of the dot operator. An expression such as $.\{500\}$ is realized as a shift register in BRAM with custom tap points for read access. No such optimization was found in the oneAPI-based kernels.

The resources used relate proportionally to the size of the NFAs accelerated on the board. Increasing the number of transitions, a side-effect of multi-character matching transformations, leads to higher resource usage. Fig. 8 further breaks down the ALUT usage of the kernels targeting the OpenCL-based kernel for the Arria 10 PAC by exposing the design methodology of the HLS compiler. Each kernel report generated by Intel's OpenCL compiler categorizes resource usage into *Feedback* and *Computation* components. *Feedback* can be loosely understood as the architecture representing, at a high level, the states and state feedback that connects the *next* (q_{i+1}) and *curr* (q_i) active state arrays. Computation encompasses the comparator logic and the selection of the set of *next* active states (q_{i+1}), defined by the NFA transitions. For smaller bandwidth kernels, the resources used to implement *Feedback* far outweigh those needed for *Computation*.

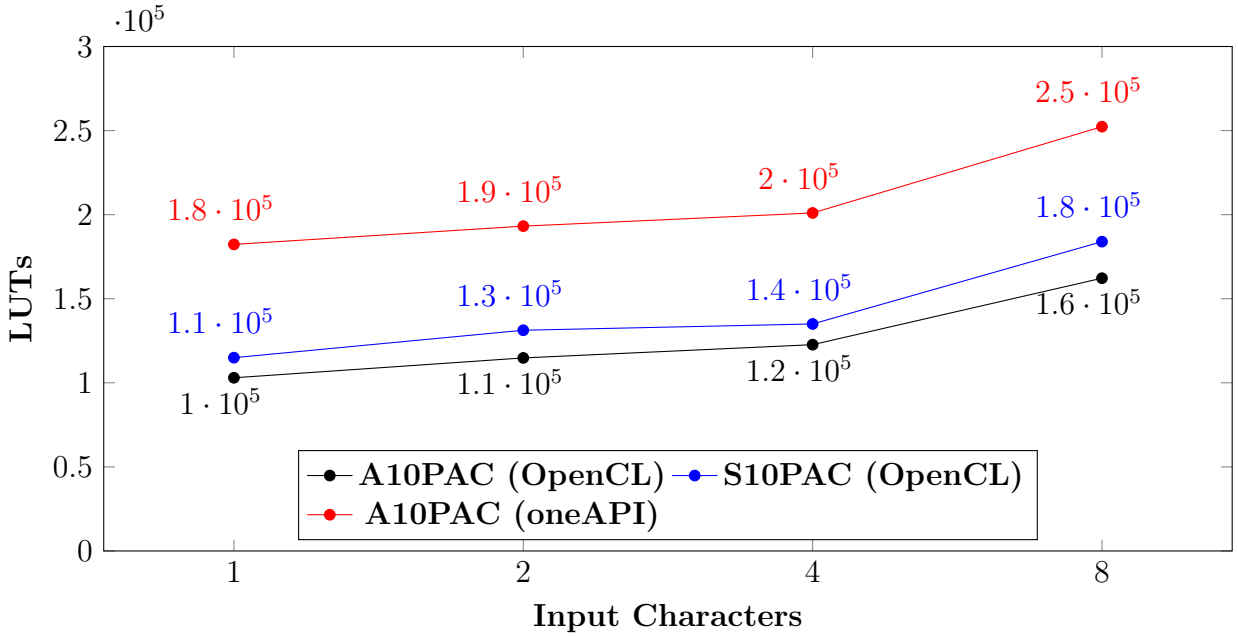

Figure 7: Total ALUT usage of OpenCL kernels with varying input bandwidths

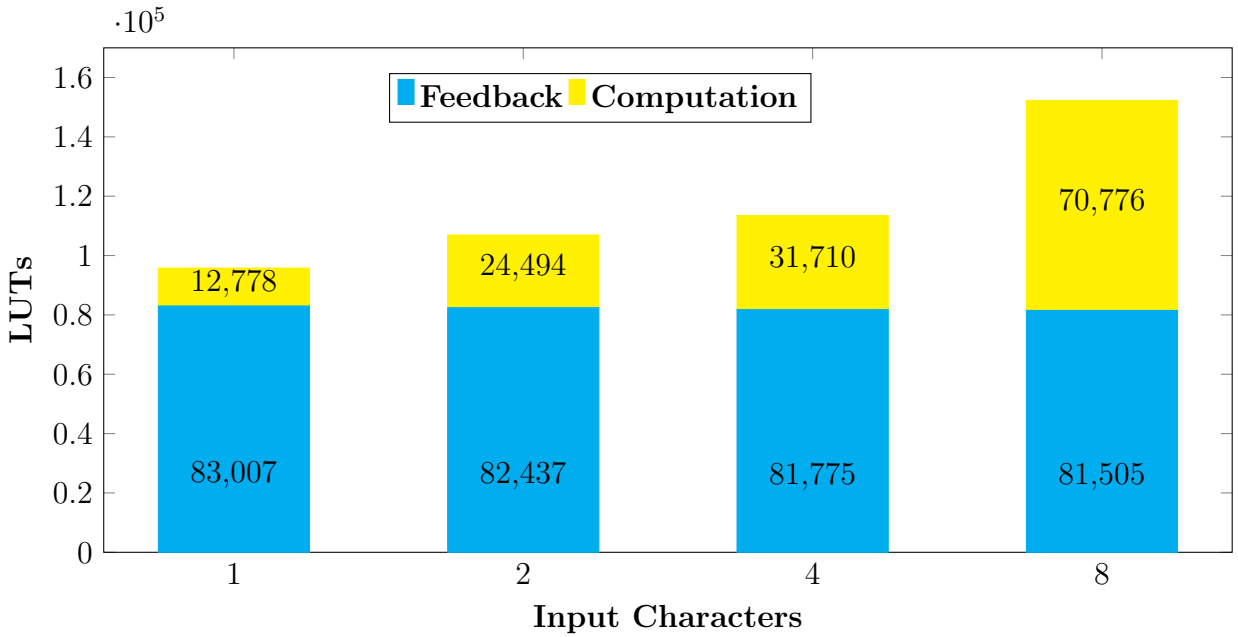

Figure 8: Breakdown of LUT usage on OpenCL kernels

Table 3: Comparison of REGEX matching engine

Design	Process Node	Input Bandwidth (bytes)	States/ Non-meta Chars	LUT/State	Max. Frequency (MHz)	Throughput (Gbps)	Throughput Efficiency
Yang, Prasanna [5]	65nm	2	**120000**	**.69**	216	3.47	5.03
Yang, Prasanna [5]	65nm	8	**120000**	1.02	160.9	10.3	**10.1**
Yamagaki, Sidhu [11]	90nm	2	7803	.81	184	2.95	3.63
Yamagaki, Sidhu [11]	90nm	8	7803	2.51	84.42	5.4	2.15
Clark, Schimmel [30]	150nm	4	17573	3.13	219	7.00	2.24
Clark, Schimmel [30]	150nm	8	17573	5.31	114.2	7.31	1.38
Our Design (A10PAC OpenCL)	20nm	2	13049	8.80	301	4.57	.519
Our Design (A10PAC OpenCL)	20nm	4	13049	9.40	295	8.77	.933
Our Design (A10PAC OpenCL)	20nm	8	13049	12.43	293	17.88	1.43
Our Design (S10PAC OpenCL)	14nm	2	13049	10.06	**331**	5.17	.514
Our Design (S10PAC OpenCL)	14nm	4	13049	10.35	306	9.56	.924
Our Design (S10PAC OpenCL)	14nm	8	13049	14.1	310	**19.38**	1.374
Our Design (A10PAC oneAPI)	20nm	2	13049	14.80	270	3.57	.241
Our Design (A10PAC oneAPI)	20nm	4	13049	15.40	263	8.38	.544
Our Design (A10PAC oneAPI)	20nm	8	13049	19.34	245	15.6	.807

7.0 Discussion

This section analyzes this work's results in regard to the needs of real-world, regex acceleration apps and compares key metrics against previous state-of-the-art designs. Design throughput, efficiency, and scalability are investigated to display the relative merits and trade-offs of this work and the use of HLS as a design tool. The OpenCL-based kernels show substantial benefits and point to the continued viability of HLS for FPGA-based designs.

7.1 Throughput

For many networking apps, the standard unit of throughput is Gbps. Therefore, all results have been expressed in this form to allow for easy comparison. The throughput measured above is the total data processed by the kernel as a whole normalized by the time to process this data, not the sum total of throughputs for all parallel NFAs. Among designs targeting similar apps and reporting throughput in the same manner, this work shows best-in-class throughput, with even the lowest performing oneAPI-based kernels exceeding previous RTL-based designs at similar input bandwidths.

While the designs of [5, 11, 30] all reach eight-character bandwidths, the maximum kernel frequency quickly degrades as the bandwidth scales, some by as much as 50% (Table 3). The maximum frequency is bounded by the time it takes signals to propagate through regions of the FPGA. More complex pipelines or larger gate latencies, caused by older FPGA fabrication processes, can play a part in reducing design frequencies. The ability of the OpenCL compiler to maintain a high frequency, coupled with more refined device fabrication resolution, gives this work a competitive edge.

7.2 Efficiency

Although throughput acceleration is the primary goal, targeting a real-world app requires some analysis of practicality. Fast packet inspection is useful only if most, or all, attacks are flagged, meaning many, or all, SNORT rules must be checked in parallel. With thousands of rules, fitting all relevant regexes becomes a challenging task. To incorporate this need, the metric of *throughput efficiency* (7.1), also called *performance*, has been used in prior works such as [5, 11, 20, 30]. This measurement incorporates the need for small, efficient NFAs and low-latency, high-bandwidth designs into a single metric for easy comparison.

$$Throughput\ Efficiency = \frac{Throughput}{LUT/State} \tag{7.1}$$

As seen in Table 3, the LUT/state of this design lags behind prior approaches, redeemed only by superior throughput. Aside from the impressive throughput efficiency of [5], which allows the authors to fit an order of magnitude more states on the board, this work's OpenCL-based results are marginally behind past RTL designs. While OpenCL maintains a high level of abstraction which limits some resource saving constructs available to RTL developers, the compilers are able to perform optimizations such as inferred shift registers and re-use of comparator logic, in part mitigating the loss of efficiency. The lack of these features, coupled with a lower maximum clock frequency, causes the oneAPI-based kernels to lag severely in this metric. Comparing the two platforms targeted by OpenCL in this work shows that the Arria PAC maintains consistently better throughput efficiency by keeping LUT usage low despite slightly worse clock speeds than the Stratix PAC.

7.3 Scalability

The scalability of this app may be investigated on two sides, throughput and expression capacity. While increasing the number of expressions searched in parallel is important for the deployment of a real-world SNORT IDS accelerator, this work's primary goal is to push the bounds of throughput. Through bandwidth scaling and a stable clock frequency, linear

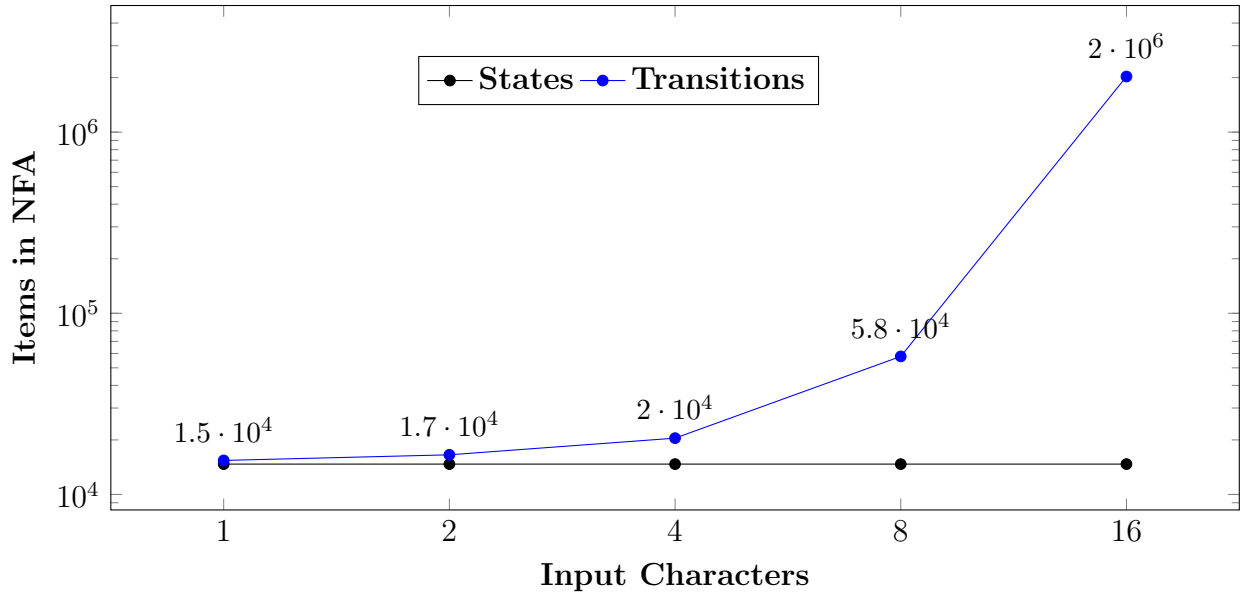

Figure 9: Transition explosion from multi-character transformations

throughput scaling is possible with up to eight input characters. The throughput efficiency also steadily improved as the throughput grows faster than the additional resources required to support more input characters. However, such scaling has been found to be unsustainable. While multi-character transformations have no effect on the number of states in an NFA, as discussed above, they do cause an exponential growth of the number of transitions between states. For lower-order (one, two, four) multi-character matching NFAs, this growth has minimal proportional impact, despite adding complexity in the form of extra transitions. The exponential effects of these transformations can be visualised in Fig. 10 and Fig. 11, the four- and eight-character matching NFA for "Pen*e". Larger input bandwidths, like the eight and sixteen character NFAs seen in Fig. 9, double the number of transitions and then leap to a 35-fold increase. Therefore, further bandwidth scaling would reduce the number of regular expressions that could fit on the board, and likely decrease the throughput efficiency as the resource needs outpace the linear gains in throughput.

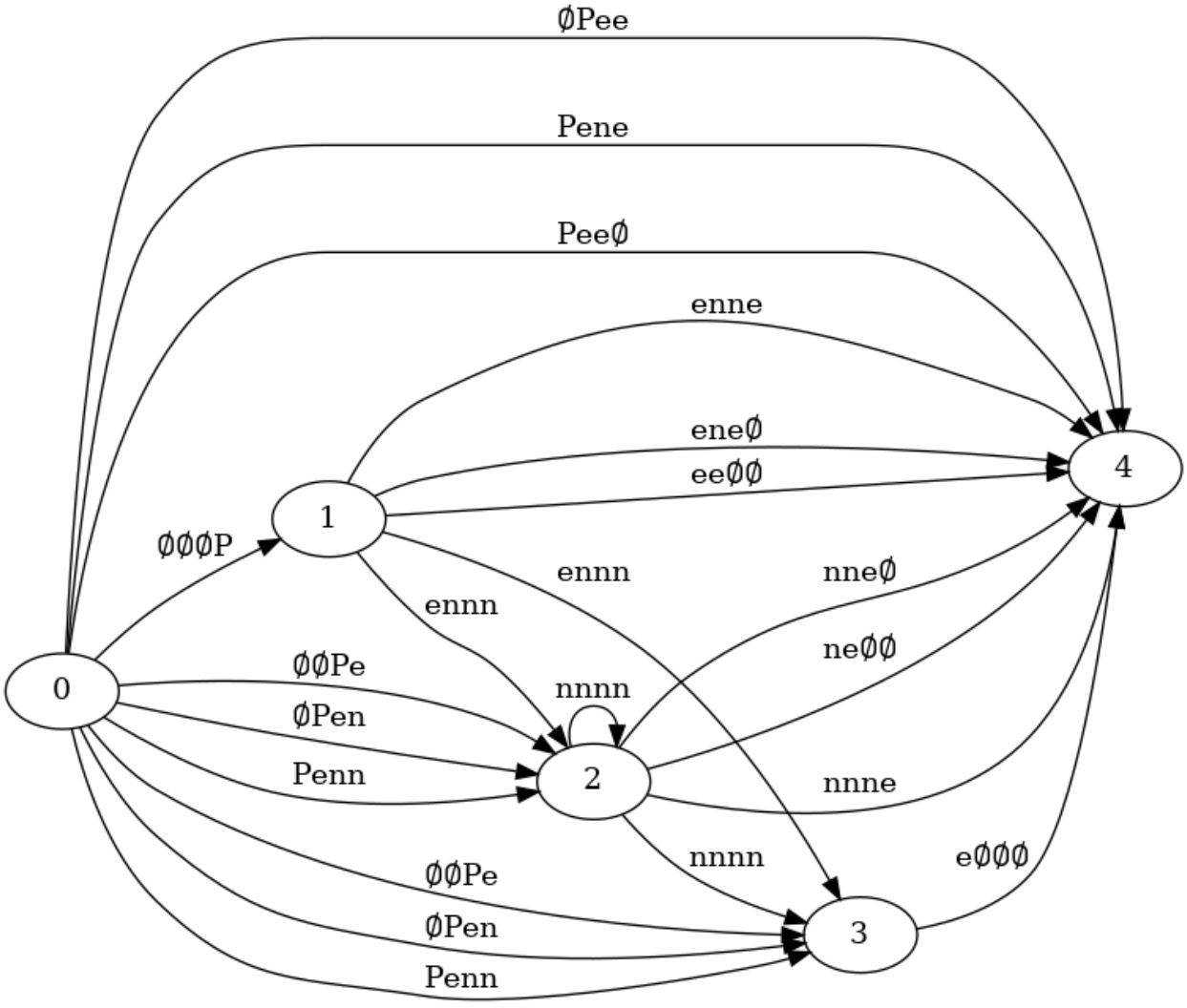

Figure 10: Four character state machine for “Pen*e”

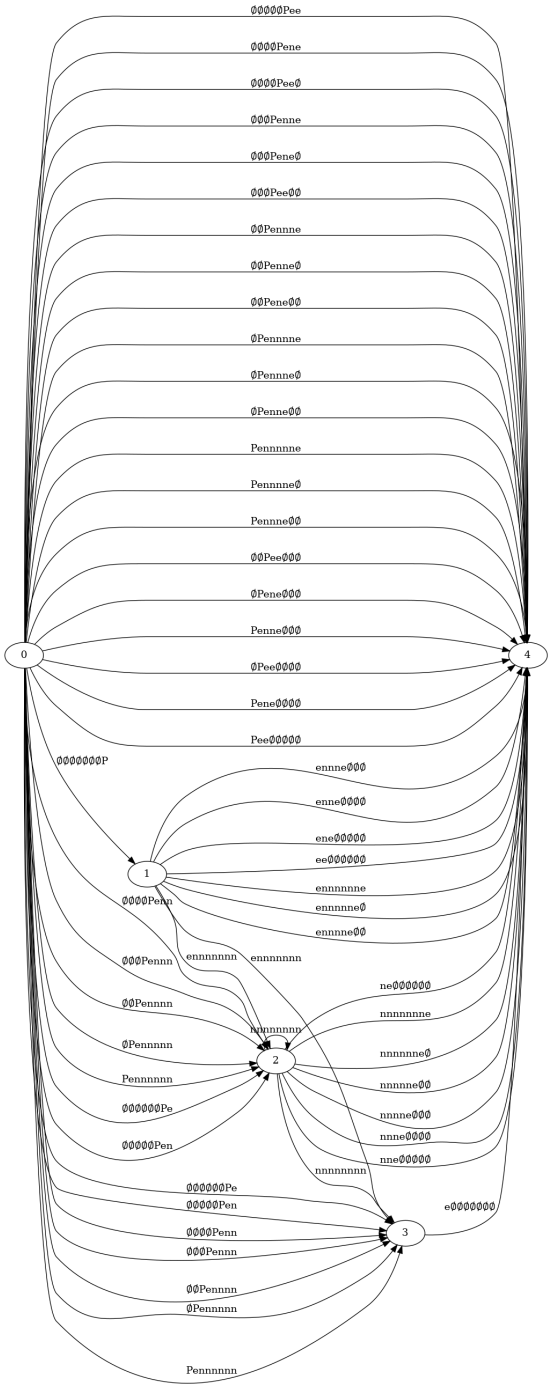

Figure 11: Eight character state machine for “Pen*e”

8.0 Conclusion

In this work, a strategy for accelerating regex searches using OpenCL- and oneAPI-based kernels on FPGAs is put forth. It is observed that the separation of the problem into memory and combinational-logic subsystems lends itself well to HLS development. The OpenCL designs for the Arria 10 PAC reached a peak throughput of 17.88 Gbps while matching eight input characters concurrently. Stratix 10 PAC designs achieve a maximum throughput of 19.4 Gbps while also matching eight input characters on the same select set of SNORT rules. Both OpenCL designs effectively scale input bandwidth while maintaining a stable design clock frequency. The high throughput allows the OpenCL-based kernels for the Arria 10 PAC to achieve a throughput efficiency near that of some previous RTL-based designs.

The newest compiler, Intel's oneAPI compiler for SYCL, may still be suffering from refinement issues as its designs performed the worst in all metrics. Despite using nearly double the LUTs in some cases as the OpenCL designs targeting the same board, oneAPI-based kernels reached a maximum of 15.6 Gbps for eight-character multi-matching. The addition of certain optimizations, such as the inference of shift registers found in the OpenCL compiler, could help alleviate the resource utilization issues found in the oneAPI compiler.

While these results are promising, investigation into NFA construction suggests that further bandwidth scaling may be impeded by exponential growth of the automaton. Doubling the input characters per transition vastly increases NFA complexity and leads to an explosion in the number of unique transitions between states.

This work finds that HLS is a valuable tool for FPGA development and can exceed the throughput of similar RTL-based designs. Furthermore, in holistic comparisons using throughput efficiency, this work shows similar results to some prior RTL-based designs. These characteristics could enable enterprises to more easily deploy network-security measures and shorten development timelines. The ability to compete with RTL-based designs, in a metric

that encompasses both throughput and resource usage, is a validation of HLS tools' ability to create effective designs and an indication that future iterations of the technology will be useful to many hardware designers.

9.0 Future Work

The oneAPI compiler signifies a significant push by Intel to improve development on a wide range of accelerators, including FPGAs. However, this toolchain is still in its infancy and is unable to compete with its more mature counterpart. The performance of both the OpenCL and oneAPI compilers must be revisited as Intel invests further into each. New generations of FPGA also provide intriguing avenues for investigation. The Intel Agilex series, fabricated on a 10 nm process and release in 2020, is one such device.

This accelerator could effectively handle 10GE networks; however, 100GE is still out of reach. Strategies to allow further bandwidth scaling while maintaining efficient use of board resources are also essential to the future of this work. One such avenue is the use of a progressive filtering pipeline, which grows in specificity at each stage. This work would likely fall into the last stage as a regex verification engine, needed to process only the small percentage of packets most likely to contain a match.

Bibliography

[1] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel J. G. van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. *Measuring the Cost of Cybercrime*, pages 265–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[2] Cisco. Snort.

[3] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 89–98, 2006.

[4] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. Infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010.

[5] Y. Yang and V. Prasanna. High-performance and compact architecture for regular expression matching on fpga. *IEEE Transactions on Computers*, 61(7):1013–1025, 2012.

[6] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[7] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, page 93–102, New York, NY, USA, 2006. Association for Computing Machinery.

[9] V. Sateesh, C. Mckeon, J. Winograd, and A. DeHon. Pipelined parallel finite automata evaluation. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 108–116, 2019.

[10] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 227–238, 2001.

[11] N Yamagaki, R Sidhu, and S Kamiya. High-speed regular expression matching engine using multi-character nfa. In *2008 International Conference on Field Programmable Logic and Applications*, pages 131–136, 2008.

[12] Le Hoang Long, Tran Trung Hieu, Vu Tan Tai, Nguyen Hoa Hung, Tran Ngoc Thinh, and Dinh Duc Anh Vu. Enhanced fpga-based architecture for regular expression matching in nids. In *ECTI-CON2010: The 2010 ECTI International Confernce on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 666–670, 2010.

[13] PCRE. Perl compatible regular expressions.

[14] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[15] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960.

[16] The Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems, Jul 2013.

[17] Intel. Intel fpga sdk for opencl pro edition: Programming guide, Jun 2020.

[18] Intel. Intel fpga sdk for opencl pro edition: Getting started guide, Jun 2020.

[19] The Khronos Group. Sycl - c single-source heterogeneous programming for acceleration offload, Jan 2014.

[20] Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2007.

[21] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. *SIGPLAN Not.*, 47(8):129–140, February 2012.

[22] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24:52–61, 2004.

[23] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. DFC: Accelerating string pattern matching for network applications. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 551–565, Santa Clara, CA, March 2016. USENIX Association.

[24] Masanori Bando, N. Sertac Artan, Rihua Wei, Xiangyi Guo, and H. Jonathan Chao. Range hash for regular expression pre-filtering. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Edsger Dijkstra. Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum, January 1961.

[26] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, page 50–59, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Jinja - modern templating for python, April 2020.

[28] M. A. Mansoori and M. R. Casu. Efficient fpga implementation of pca algorithm for large data using high level synthesis. In *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, pages 65–68, 2019.

[29] Project Gutenberg Literary Archive Foundation. Project gutenberg.

[30] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257, 2004.