ECTI Transactions on Computer and Information Technology
Journal homepage: https://ph01.tci-thaijo.org/index.php/ecticit/
Published by the ECTI Association, Thailand, ISSN: 2286-9131

Streaming Accelerator Design for Regular Expression on CPU+FPGA Embedded System

Hendarmawan[1], Morihiro Kuga[2] and Masahiro Iida[3]

ABSTRACT

A pattern matching application is one of the necessary tasks in streaming data processing. A hardware accelerator employing FPGA can be faster than a general-purpose processor in performing data pattern matching using regular expression methods. However, it is difficult and time-consuming to design the hardware on the FPGA for each regular expression pattern. We are researching a method for automatically designing hardware accelerators for higher efficiency and performance to improve user productivity. In this research, we propose rules and methods for translating regular expression patterns into supported hardware code as our contribution to providing an efficient design method for regular expression hardware accelerators and allowing the efficient utilization of FPGAs. The performance evaluation is compared with the regular expression algorithm on ARM processors, CPU servers, and FPGA data streaming applications. Our result shows that our FPGA accelerator enables speeding up data streaming applications on CPU processors. Our solution is 733 times faster than optimized C/C++ code. It is 70 times faster than using the Python library. It is twice as fast as PYNQ-Z2 and 1.5 faster than RE2C. Furthermore, our proposed accelerator Ultra-96 improves the performance 2 times with an 8[MB/J] high energy efficiency from the previous PYNQ-Z2 approach.

Article information:

Keywords: Stream Processing, Pattern Matching, Complex Event Processing, Hardware Accelerator

Article history:
Received: May 16, 2021
Revised: July 4, 2022
Accepted: July 30, 2022
Published: October 15, 2022 (Online)

DOI: 10.37936/ecti-cit.2022164.249268

1. INTRODUCTION

Real-time data stream processing in edge computing has become a priority for data analysis of stored big data. Data aggregation and processing algorithms like pattern recognition, complex event processing (CEP), and high-speed data processing for massive sensing of data for real-time processing are becoming a must [1]. Regular expression (regex) pattern matching and feature extractions are the core of big data analytics and processing real-time data into meaningful information [2].

Because of high productivity with rich libraries and support, data scientists use Python as the high-level programming language to perform these processes [3]. However, its performance is slow, and power consumption is also high. Therefore, it is necessary to employ a Field Programmable Gate Array (FPGA) accelerator to speed up and scale-up performance and, at the same time, lower the energy consumption [4].

There are two main challenges in integrating FPGAs into the edge computing environment: the high Hardware (HW) and Software (SW) co-design complexity, and HW resource sharing. First, HW and SW co-design for developing an accelerator requires developers to have expertise on both sides, especially for significant architectures and complex projects. High-Level Synthesis (HLS) methodology helps in these scenarios. However, HLS is under development with many limitations and requirements [5].

Second, centralized computing servers are highly virtualized. Multiple applications can be concurrently executed without disturbing each other, which improves the server utilization rates but leaves the risk of data overhead while processing data from the IoT ecosystem. To utilize FPGAs in the same manner, it is necessary to provide virtualization techniques for sharing the resources of many FPGAs. Useful techniques include lookup tables (LUTs), block RAMs (BRAMs), and digital signal processors

[1,2,3] The authors are with Faculty of Advanced Science and Technology, Kumamoto University, Kumamoto-shi, 860–8555 Japan., E-mail: hendarmawan@st.cs.kumamoto-u.ac.jp, kuga@cs.kumamoto-u.ac.jp and iida@cs.kumamoto-u.ac.jp
[1]Corresponding author: hendarmawan@st.cs.kumamoto-u.ac.jp

(DSPs). These allow real-time processing accelerators to be implemented.

This paper proposes rules for designing regex HW Intellectual Property (IP) for FPGA Accelerators. We also investigate the behavior and evaluate the circuit scale and energy consumption. Our approaches works well on stream data processing related to regexes.

The rest of this paper is organized in five sections. In Section 2, we summarize the state of the art from past research. We introduce the architecture of the proposed FPGA accelerator, streaming application, rules, and data streaming framework in Section 3. Section 4 provides the details of using the proposed translation rules for streaming data analysis, building accelerators, and optimization techniques. Implementation details and experimental results are provided in Section 5. Finally, Section 6 concludes the paper.

2. RELATED WORK

Traditional recommendation algorithms are usually classified into two types: collaborative recommendation and content-based recommendation [5]. Collaborative recommendation uses the past behaviors of several users to produce new recommendations. The limitation of this type is that a large history of past behavior of users is required to make a good recommendation. Moreover, scalability problems can be encountered because all preferences of all users must be collected, which results in sparse and excessive data. Examples of this algorithm are matrix factorization and the restricted Boltzmann machine.

Content-based recommendation analyzes the property of each item and makes a recommendation based on the user's profile and past behaviors. The limitation of this algorithm is that it cannot recommend new items apart from items related to the user's past behaviors. The examples of this algorithm are naïve Bayes, decision tree, k-nearest neighbor (KNN), and support vector machines. Collaborative recommendation and content-based recommendation both have unique benefits and disadvantages.

2.1 Related Work

Big data processing requires processing a high volume of event streams in real-time. Examples of this kind of processing include financial systems, stock exchanges, network surveillance, and health care. These all requires processing stream events in real-time [6]. Software-based stream monitoring has limitations due to high network packet rates, as shown in [7]. Event Processing Hardware using FPGA accelerator has been implemented by ETH Zurich University on publication [8], achieving high-performance regex engine over data stream by which they have developed using VHDL / low-level programming language. Using native design flow will enable boost performance; however, it has drawbacks on low productivity as it requires many years to develop their pattern matching hardware accelerator due to complexity.

FPGA has demonstrated excellent speed performance and power efficiency advantages over conventional computers in various domains, such as image processing, communication, and data analysis. In reality, the design and implementation of hardware on FPGA typically take a more extended time due to its complexity for core development. It requires understanding and skills for circuit design to use tools to develop accelerator applications on FPGA. This disadvantageous attribute makes the development cost of FPGA expensive.

A pattern matching and feature extraction engines are mainly implement regex algorithms. Regex matching is an important mechanism used by popular network intrusion detection systems (NIDS), such as Bro [9] and Snort to perform deep packet inspection against potential threats. There are few implementations of software-based regex for security, like research by [10] to perform SNORT detection [11]. For hardware-based implementations for pattern matching and stream processing feature extraction has been carried out [12] to develop regex engines for hardware circuits using the hardware description language VHDL. In [13], a design, implementation, and evaluation of a high-performance architecture for pattern matching are performed on FPGA to counteract an increasing number of patterns to be scanned and network bottleneck. The main objective of using FPGA to solve this problem is to accelerate the system to achieve high performance.

3. PROPOSED RULES FOR FPGA PATTERN MATCHING

3.1 Problem and Challenges

This research improves upon prior research on data processing pattern matching using regex on processors and hardware FPGA. The original objectives of the authors were investigate different approaches for high-level programming libraries like C/C++ and Python for pattern matching on software and how to improve the performance over traditional pattern-matching software using hardware accelerators. We also wanted to improve productivity by designing an automation building block for process for software and hardware developers. Each development flows from a given pattern to code generation using high-level programming language into hardware logic abstraction, design synthesis, and implementation, and finally generating a bit-stream for hardware overlays for hardware accelerator. We propose a hardware design and abstraction techniques to optimize and utilize hardware resources to get better results and evaluations.

There are problems and challenges to overcome to achieving our research's objectives. First, we need to learn how to get better performance over traditional pattern-matching software with lower power consumption. Second, we must investigate how to promote co-design spaces for software and hardware developers by designing automation building blocks for process so that each development flows. Third, we want to encourage resource sharing within a FPGA to optimize and utilize hardware resources. Problems and challenges include:

1. The development flow of the FPGA applications needs complex and complicated hardware expertise. However, they guarantee better performance and lower energy consumption. The initial challenge for hardware developers required complicated coding algorithms and hand wiring blocks of programmable array logics from bottom-up to present IP. Finally developers must use shells of a designated vendor FPGA to implement their code at the HW level. These bottom-up principles are challenging and often become obstacles for software developers who want to design and deploy their applications at the HW level.
2. The complexity of each development process at the HW layer is time-consuming and these steps are contrary to rapid development scenarios for high productivity principles. Therefore, some changes are required to simplify and automate the process.
3. Pattern matching software on processors is easy to use with libraries provided by a high-level programming language like C/C++ and Python. However, the performance of these implementations is commonly lower compared to complex coded pattern-matching software. Meanwhile, pattern matching and feature extraction on general purpose processors have lower performances and higher energy consumption than HW in a custom chip implementation.

3.2 Developing Pattern Matching

The majority of High-level programming languages like C/C++, Python, and Java offer extensive libraries to help programmers get easy-to-use functions and procedures to implement their algorithms and applications. For pattern matching algorithm, C/C++ provides a regex library, and Python provides re library, which can be quickly deployed by importing this library into their code. There are advantages and disadvantages either using software or hardware for programmers to perform their pattern matching algorithm in terms of productivity, computing performance, difficulties, limitation, complexity, and energy consumption.

Programmers have the freedom to express their way of thinking and problem-solving techniques in their algorithm based on software principles. However, it is a huge challenge to achieve a rapid development process, solving compatibility issues, trouble-shoot, minify resource allocation, and obtain peak performances for hardware implementation like FPGA, which require high amount of skill and time to make them perform with high performance and low energy consumption [14].

Hardware developers and vendors introduced HLS [15], a state-of-art C-to-FPGA synthesis solution, to improve design productivity. HLS helps programmers to implement algorithms using C language. Then HLS translates it into HDL like VHDL or Verilog. However, the C libraries cannot be used on HLS due to limitations described in the HLS manual. Therefore, the author employs RE2C to implement regex pattern matching on the hardware level because RE2C is DFA-based and suitable for programming logic for a FPGA. Our proposed method combines high productivity from RE2C and HLS approaches, enabling rapid development for both hardware and software developers who want to implement regex pattern matching accelerators in hardware.

3.3 Overview of RE2C Code generator

Regular expression to C (RE2C) is a free and open-source software lexer generator for C. Originally written by Peter Bumbulis and described in his paper [16]. It is the lexer generator adopted by projects such as PHP, Spam Assassin, the ninja build system, and others. The problems and challenges of using regular expression patterns for data stream applications are as include:

1. Limited support for hardware implementation.
2. Slow performance when using the C Library for Regex.
3. Scarce guidelines for beginners and hardware accelerators.
4. Low productivity with hardcoding necessary for each step, which makes for HW-SW co-design slow and tedious.

Unlike other generators, it does not provide default rules, input pseudo-token, and buffer management routines [17] which must instead be provided by users.

3.4 Internal Process of RE2C

Pattern matching using RE2C begins with scanner getting input of regex then passing it to the parser generator, then it is further processed in semantic analyzer before it optimized and generated into C++ code is generated. Its simplified rules are:

1. Constructing DFA
   The first step of the RE2C scanner generator is constructing a DFA from the pattern to recognize the implemented regex provided by the user.
2. Generating Code
   After constructing DFA, the following steps are parser generator, semantic analyzer generator, optimizer, and code generator. Because of using DFA, code generated by RE2C is relatively

straightforward. It will create some additional code to save backtracking information.

3. Buffering
The RE2C generated scanner will check if a buffer is needed by comparing YY-CURSOR and TT-LIMIT. This attempt is performed to reduce the amount of checking. This will minimize the steps needed for checking every running routine.

RE2C generates minimalistic hardcoded C or C++ DFA state machines to process regular expression syntax input. After that, we directly compile and use the generated C code for software evaluation. However, modification and hardcoding based on the RE2C generated output C code are required to be able to run in the HLS due to compatibility and support issues. We propose some rules are required to make HLS compatible code in the next section.

4. TRANSLATION RULES OF REGULAR EXPRESSION FOR HARDWARE ACCELERATOR

We proposed special rules for regex translation for FPGA hardware accelerators targeting high-efficiency, and low-cost HW/SW co-design. This leads to shorter development curve and development time because we can reduce development complexity and the language barrier between high-level and low-level programming languages at the HW level. This can be achieved by implementing code translation and linker for HW-supported codes. After a series of experiments, we developed a modification of RE2C together with the HLS technique to develop accelerators quickly with attractive acceleration performance. In addition, some optimization is also provided to avoid data overhead and to perform pipelined data operation.

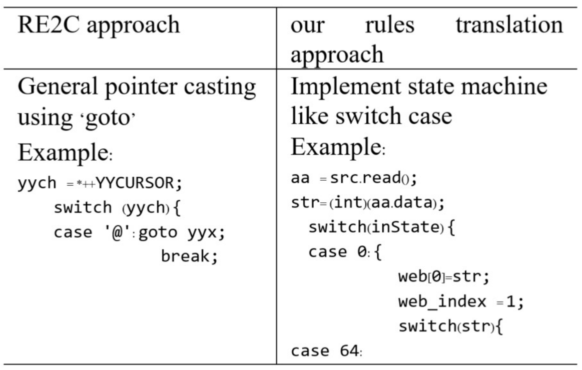
4.1 Into HLS C

Vivado HLS [21] is one of the HLS tools by Xilinx to bridge the hardware and software domains. It helps hardware developers to work at the level of abstraction while creating high-performance hardware. Meanwhile, for software developers, it has tools to accelerate computation for their algorithms on FPGAs. HLS allows them to develop an algorithm at the C-Level rather than hard-coding in low-level programming hardware languages like what Hardware programmers do in HDL languages like Verilog or VHDL. Furthermore, HLS allows users to verify code and function more quickly than traditional DHL. It allows control C-Synthesis through optimization directives and makes it possible to create multiple implementations for different purposes.

4.2 Proposed Translation Rules

There are limitations on Vivado HLS C supported code [21]; it does not support dynamic memory allocation, OS operations, or general pointer casting. Therefore, this paper proposed translation rules to solve these limitations by modifying RE2C to create C compatible standard HLS. These rules are:

1. **Avoid Dynamic Memory Allocation**
With Static Memory Allocation, users are required to declare variables before using them so that the compiler can allocate these variables to the memory. On the other hand, Dynamic Memory Allocation does not require the user to specify the memory allocation required for the program in advance users do not need to worry about any upper limit for memory allocations. These advanced features are not supported on HLS. Thus, the first rule is to adjust the lexical analysis routines in RE2C to avoid dynamic memory allocation use.
2. **Directing Operating System Operations to caller**
Operating System (OS) operations such as file read or write and OS queries like time and date are not supported by HLS. Therefore, the second rule for translation is managing OS operation into an outer platform like a driver and test bench in C/C++. With OS direct call restriction, all data from and to FPGA must be read and written from the input and output ports, respectively.
3. **Changing General Pointer Casting into State Machine**
C-style pointer casts, in this case, are "goto" statements, which are not supported by HLS. These limitations exist probably because, first, it can be challenging to manage flow control. Second, it may be error-prone, which can cause disaster (wild pointer) if used excessively. Therefore, in the third rule, we replaced this pointer casting with a simple state machine which works perfectly, and more importantly, it is supported by HLS. RE2C is considered the fastest framework for regex pattern matching compared to optimized C and Python regex libraries and code generators for re-patterns into C. Our proposal enables accelerating its computation at the hardware level with our proposed rules.



RE2C approach	our rules translation approach

General pointer casting using ‧goto‧
Example:
`yych =*++YYCURSOR;`
`    switch (yych){`
`    case '@':goto yyx;`
`                break;`
	Implement state machine like switch case
	Example:
	`aa =src.read();`
	`str=(int)(aa.data);`
	`  switch(inState){`
	`  case 0:{`
	`            web[0]=str;`
	`            web_index =1;`
	`            switch(str){`
	`case 64:`

4. **Inserting design directives and wrapping code**
Our translator translates the regex function generated by RE2C for High-level synthesizing with

Vivado HLS. It contains pre-developed data structures, data processing, and common functions that can be reused in various regex applications. Adding HLS directives and wrapping code will provide IPs and drivers to accelerate the functions of the designed regex application. Vivado HLS can support original SW classes and HW versions at the runtime after translation. Therefore, applications can be accelerated without any modifications on Vivado HLS. Although this method can only accelerate regex functions imported into applications, it is meaningful because SW engineers can easily use the regex HW class library generated by Vivado HLS.

Our approaches are the best fit for stream data processing related to regular expressions. However, there are currently some limitations. First, not all regular expression matching has been tested. Second, these rules and translation do not support a complex regular expression requiring resource sharing, much backtracking, and a deep first algorithm.

4.3 Regex Accelerator

It is easier to develop HLS-compatible C/C++ code by benefiting from our proposed rules. This is crucial for designing regex HW accelerators, and any regex pattern would be supported as long as it is a standard C code reference supports it.

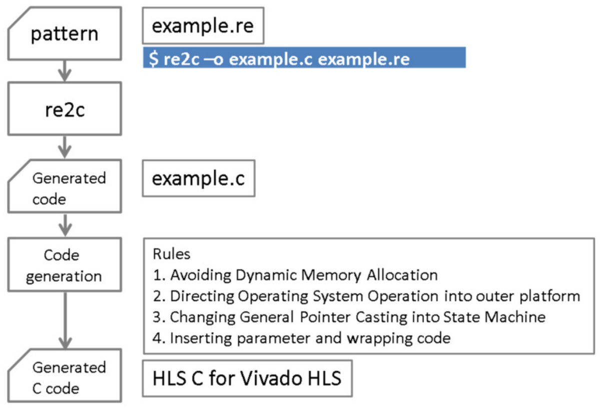

pattern
example.re
$ re2c –o example.c example.re
re2c
Generated code
example.c
Code generation
Rules
1. Avoiding Dynamic Memory Allocation
2. Directing Operating System Operation into outer platform
3. Changing General Pointer Casting into State Machine
4. Inserting parameter and wrapping code
Generated C code
HLS C for Vivado HLS

Fig. 1: *Development of data streaming accelerator workflow.*

Figure 1 shows a workflow for processing regex pattern using an algorithm suitable for enabling high-performance HW on an FPGA accelerator. It start with a regex pattern, then RE2C generates generic C++ code with the command "re2c -0 example.c example.re" where "example.re" is the input code, and "example.c" is the output code. Then, using our translation rules, we generate C++ code for HLS to design HW accelerator IP. Finally, regex HW is generated by Vivado HLS and Vivado for the PYNQ devices.

Source code 1 - Accelerated HW on HLS

```
extern "C" {
void ip_accel(hls::stream<int_s>& src, hls::stream<int_s>&
dst){
 #pragma HLS INTERFACE axis port =src
 #pragma HLS INTERFACE axis port =dst
 #pragma HLS INTERFACE s_axilite port =return
 #pragma HLS dataflow
 int A[N];
 int C[N];
    axis2Arr(src, A); //convert AXI -> Array for PS-PL
    accel_hw(A, C); //function accelerated HW
    Arr2axis(C, dst); //convert Array -> AXI for PL-PS
 }
}
```

Source code 1 shows HW interface design to connect with regex "accel_hw" it works as follows: Function "*ip_accel*" handles data streaming communication with "HLS INTERFACE axis port" from axis source, using "HLS INTERFACE axis port" destination for results, and "HLS INTERFACE s_axilite" for the controller. Then function "$axis2Arr()$" converts axis data into array data A that we can process reading all streaming and converting it in real-time. Function "$accel\_hw()$" is the function which runs the regex pattern, where all streaming data on array A is processed with a Pipeline DFA State machine. As a result, every matched character is be saved and then transferred into array C. Finally, Function "$Arr2axis()$" converts data from array C into the AXIS type, channeled to the "HLS INTERFACE axis port" destination.

4.4 Design Parameter on Vivado HLS

HW performance can be improved by using the design parameter offered by HLS compilers. These parameters aim to improve HW performance, and HLS directives are inserted onto the C program as pragmas. We incorporate the design parameter on source code 1 as pragma dataflow, pragma HLS pipeline, pragma HLS unroll and pragma HLS loop_flatten off in the accel_hw function.

The following three are complementary technique to our work to allow using optimized hardware FPGA:

A. Loop_flatten off
   Loop flattening unrolls instructions inside a loop, to remove the overhead associated with checking for the end of the loop at the beginning of each iteration.

B. Loop unroll
   The iteration loops of the HW implementation are unrolled into independent mechanisms instead of one for parallel computation of FPGA regex accelerator.

C. Pipeline Technique
   Pipelines are a scheduling technique used in HLS. Pipeline execution allows concurrent execution of operations within a function by allowing the sub-

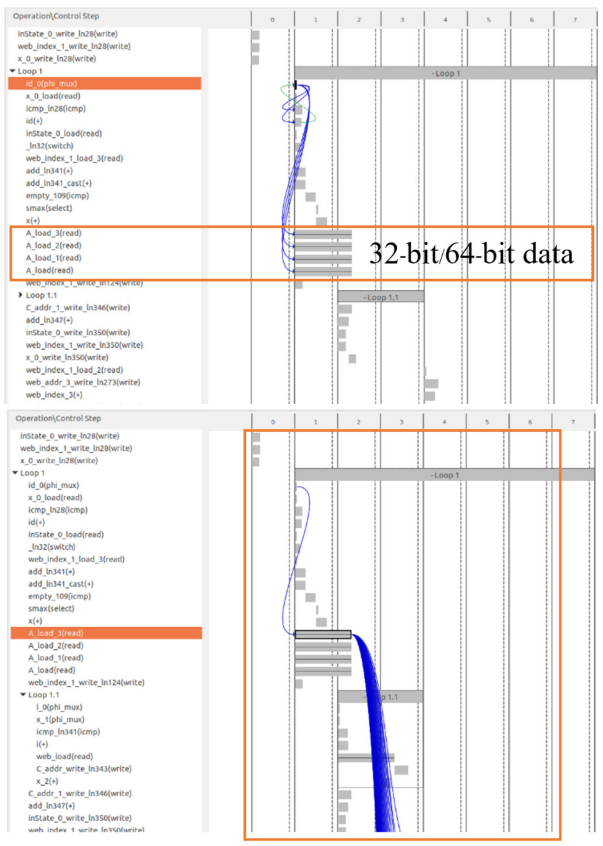


Fig.2: *Internal behavior and structure of proposed regex HW.*

sequent execution of the task to begin before the current execution has been completed. For our improved implementation, the operation between stages can be well pipelined with an interval of 1, which means our HW can process one incoming key in each clock cycle. This behavior are illustrated in figure 2. It is explained data workflow for proposed regex HW where data pipelined into four channels of AXI-Array A data read on our "*accel_hw*()" function (function accelerated HW on our source code 1) in 32 bit for Pynq-Z2 and 64 bit for Ultra96. It is further processed into DFA on our Intellectual Property (IP) on the HW level. This behavior explained the internal technique of our rules to accommodate faster processing regex pattern matching in real-time uses, which is compatible with the FPGA accelerator, where it can be done at the software level.

5. EVALUATION

We implemented pattern matching using regexes both software and hardware approach for implementation and evaluation. Our evaluation focuses on the novelty use of our rule framework for cross-platform regex computation. We repurpose existing hardware on the PYNQ Family (Pynq Z2 and Ultra96) for debugging and do not introduce additional overhead. We are able to reduce overheads with our rules, offering novelty on rapid transition and translation between Software (SW) and Hardware (HW) accelerator for streaming data computation with regex without sacrificing performance.

Table 1: *Hardware details.*

Features	CPU	ARM+ FPGA	ARM+ FPGA
Vendor	Intel CPU	Tul PYNQ-Z2	Avnet Ultra96v2
Processor	Core i7	Arm Cortex-A9	Cortex-A53
Cores(threads)	6 (12)	2	4
Architecture	64 bit	32 bit	64 bit
Process	32nm	28nm	28nm
Clock Freq.	3.4 GHz	450 MHz	450 MHz
Level 1 cache	256 KB	32 kB	32 kB
Level 2 cache	1 MB	512 kB	1MB
Level 3 cache	8 MB	-	-
TDP	130 W	4 W	24 W
Memory	16 GB	512 MB	2GB
OS	Ubuntu 18.04 LTS	Ubuntu18.04 LTS	Ubuntu18.04 LTS
PS-PL max. data transfer	-	800MB/s	1600MB/s

The system setups are shown in table 1. In the implementation, we use four different setups: System CPU Processor using Intel Core i7 servers with DDR3 memory 16GB. Second, we employ ARM on PYNQ Z2 which has a 650MHz dual-core Cortex-A9 ARM type processor with 512MB of DDR3 memory. Third, we used a ZYNQ XC7Z020-1CLG400C on PYNQ Z2 boards. Fourth, we used a Xilinx ZYNQ UltraScale+ MPSoC ZU3EG A484 with Micron 2 GB LPDDR4 memory on Ultra 96 Board. Pynq is an open-source project from Xilinx that makes it easy to design embedded systems with Zynq Systems on Chips [18].

5.1 Dataset

The email dataset contains approximately 500,000 emails generated by employees of the Enron Corporation. The Federal Energy Regulatory Commission discovered it during their investigation of Enron's collapse between 1999 and 2003. We used the May 7th, 2015, dataset version [19]. We divided into ten different sets of datasets for the evaluation.

5.2 Case Study and Implementation

We use a case study for the most common uses of regular expression pattern matching used in real-world problems, as explained in table 2. These five case studies are Email addresses, URL addresses, ZIP codes (US), Phone numbers, and Date calendar regex pattern matching, relevant for the data in the dataset. Other regex patterns also can be applied. However, we only focused on these five. From these patterns, we then perform pattern matching using different scenarios and environments to understand better the relationship between patterns, regex libraries on C/C++, and Python, and the performance for these scenarios.

Table 2: *Configuration of a Hadoop cluster.*

No	Use Case	Regex pattern
1	Email address	[\w.%+-]+)@([\w.-]+\.[a-zA-Z]
2	URL address	[www]+\.[\w.-]+\.[a-zA-Z]
3	Zip code (US)	\s+[A-Z]{2} +\d{5}
4	Phone number	[(][\d]{3}[)][ ]?[\d]{3}-[\d]{4}
5	Date	d\d\s(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s\d{4}\s\d{2}:\d{2}

The measurement of code productivity is based on a different approach to the workflow than the one used normally in this research. It measures the total number of instructions in each step of the workflow for the initial re code as input to RE2C, C code as generated code from RE2C, than for the HLS C code after translation of our rules with optimization methodology, and finally for the Verilog HDL of RTL generated code after HLS.

Table 3 shows the total number of lines of instructions for five different case studies (Email address, URL, ZIP Code, Phone Number, and Date Calendar). Input code "*.re" are identic of all case studies because there are no different codes except the regex pattern string. Translated C codes from "*.re" by re2c are required to modify for Vivado HLS using our proposed rules. It takes some costs to rewrite by hand rewriting and may lead to some bugs. Using our translation tool with proposed rules omits the hand rewriting cost and highly efficient.

Table 3: *The number of lines of code for various case studies.*

	Email	URL	ZIP	Phone	Date
Re code re2c	49	49	49	49	49
C code from re	595	471	216	263	427
Translated Code	**382**	**211**	**309**	**170**	**352**
Verilog HDL	1,380	860	923	898	957

Implementation details can be seen in Table 4. For regex stream data processing with the five case studies mentioned previously, we implemented application design with eight different architecture approaches implemented on CPU server, ARM processor, and proposed FPGA evaluation on PYNQ Z2 and Ultra96 boards. We evaluate the performance of C/C++ optimized Library, Python Library, and RE2C on both CPU and ARM processors compared to FPGA accelerators.

5.3 Architecture design on PYNQ

Our architecture for hardware regex accelerators is designed with the FPGA design Tool Vivado/Vivado HLS version 2019.1(see Figure 3). It begins with the data stream passing through PS, using C/C++ code and Python API on PYNQ Framework. The data is then sent to a memory map at the Processing System (PS) level. Passing through a Python kernel and shell driver, the data in the memory map is connected to Direct Memory Access (DMA) on the hardware FPGA. Then the data is processed on Intellectual Property (IP) on Programming Logic (PL) level. Communication between the DMA and custom IP using AXI Stream with 100MHz for PYNQ-Z2 and 187MHz for Ultra96 was the operating frequency and 32bit/64bit data widths. The generated result from regex IP on the PL side is then transported with DMA one more to be exported to the PS side using a memory buffer. Finally, the resulting pattern matching and feature extraction instructions are delivered to an application like Jupyter Notebook for interactive use.

Table 4: *Implementation methods for evaluation.*

Type	Methods
A1	C Library for CPU on PC
A2	C Library for ARM on PYNQ
B1	Python Library for CPU on PC
B2	Python Library for ARM on PYNQ
C1	RE2C for CPU on PC
C2	RE2C for ARM on PYNQ
D	Proposed FPGA on PYNQ Z2
E	Proposed FPGA on PYNQ ULTRA 96

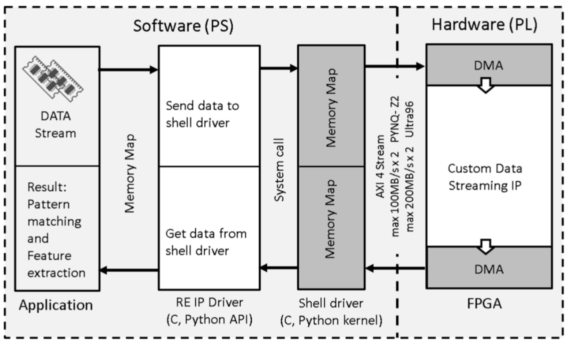


Fig.3: *Architecture design on PYNQ Framework.*

5.4 Result

The implementation result can be seen in Table 6, where all cases of pattern matching with ten different datasets implemented on CPU, ARM, and FPGA. Based on our evaluation, regex pattern matching on ARM or CPU servers using C/C++ and Python Library is much slower due to type checking and other overhead needed to interpret code and support C/C++ and Python abstractions. Sometimes, the user faults must also be taken into consideration. That case is called Catastrophic Backtracking, which requires recursively backtracking for finishing a line/dataset. Furthermore, the time complexity both of the C++ and Python Library, which uses NFA, leads to overhead. Instead of having $O(N * M)$ complexity in terms of the number of states N and transitions M of the input NFA, they can have $O(2M)$ or worse. The evaluation C/C++ regex library is slower than Python re library because, in the current *std* :: *regex* design and implementation, the regex pattern is parsed and compiled at runtime even though it does not need a runtime regex parser engine as the pattern is known during compilation in many use cases. Meanwhile, RE2C and our approach to translating NFA into DFA mean pattern matching can be processed with linear time complexity ($O(N)$ complexity) resulting in higher performance.

Finally, using our rules to adapt C++ on HLS is resulting ultrafast regex accelerators. Our approach using optimized, pre-compiled C code is able to avoid a lot of the overhead. On the HLS side, we implement optimization using data flow, pipeline, and loop unrolling. Another reason why our approach using hardware accelerators is fast is the use of memory map and DMA to transport streaming data. Then AXI 4 Stream custom IP does the computation.

The overall performance of our proposed accelerator consists of five different case studies and ten different datasets sizes. Email, ZIP, URL, Date calendar, and Phone regexes have similar profiles, with throughput distributed fairly evenly throughout the evaluation. While Data increases lead to throughput increases towards the end of the dataset. All five groups also show a peak of around 18-24% of maximum throughput at 47.26MB/s compared to 200MB/s in theory [20]. Evaluation on Ultra96 achieved 17-24% of maximum throughput, which is almost double compared to PYNQ Z2's result at 88.57MB/s compared to 375MB/s in theory. Our lightweight hardware IP is able us to perform at a maximum 187.5MHz frequency clock based on HLS runtime and evaluation. The reason why the measured throughput is only one-fourth of the theoretical peak performance is the IP can only process one character for each clock.

Figure 4 illustrates the speed-up of performance when comparing our FPGA accelerator to CPU approaches. Figure 5 compares the speedup between FPGA and ARM's performance . Implementation using RE2C for software processors on CPU and ARM is quite fast compared to equal benchmarks with C/C++ and Python regex library toward evaluation, whereas PYNQ-Z2 and Ultra96 accelerators are faster among all other evaluation both CPU and ARM processors.

The hardware accelerator using our rules is shown in Appendix 2. It dominates performance up to over 3,900 times that of A2 (C/C++ Library ARM), up to 418 times that of B2 (Python Library on ARM), up to 275 times that of A1 (C/C++ Library CPU) and 39 times that of B1 (Python Library CPU). Meanwhile, compared to RE2C, our implementation also dominated up to 39 times that of C2 and processing up to 1,4 times that of C1 despite the low hardware specifications and low power consumption of the Ultra96 board.

We also evaluated head-to-head with the PYNQ-Z2 Zynq programmable device, which is significant. Improve by doubling the performance for all case studies and datasets.

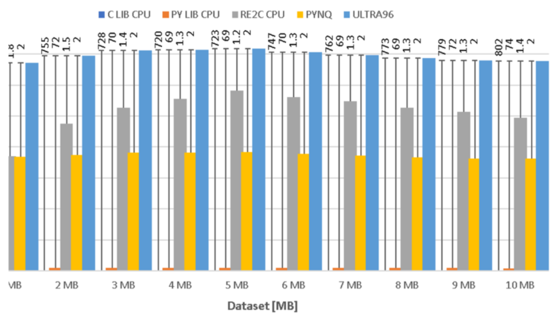


Fig.4: *Speedup of performance in comparison between PYNQ FPGA Accelerator vs. CPU approach.*

Table 5: *Resource utilization of FPGA.*

		LUT	FF	BRAM
PYNQ Z2	Email	1,937 (3%)	514 (~1%)	128 (45%)
	URL	840 (1%)	478 (~1%)	128 (45%)
	ZIP	1,015 (1%)	491 (~1%)	128 (45%)
	Phone	900 (1%)	482 (~1%)	128 (45%)
	Date	1,050 (1%)	493 (~1%)	128 (45%)
ULTRA 96	Email	2052 (2%)	414 (~1%)	108 (25%)
	URL	849 (1%)	414 (~1%)	108 (25%)
	ZIP	1022 (1%)	414 (~1%)	108 (25%)
	Phone	911 (1%)	414 (~1%)	108 (25%)
	Date	1056 (1%)	416 (~1%)	108 (25%)

Resource Utilization

The fundamental building blocks inside of an FPGA are the flip-flop (FF), the lookup table (LUT), and the Block of RAM (BRAM). Table 5 shows the resource utilization for three window parameters, Flip-flop (FF), Look Up Table (LUT), and Block of RAM (BRAM) for the PYNQ-Z2 and Ultra96 FPGA boards. The resource utilization number is provided by the Vivado tool. In contrast, LUT, FF, and BRAM are the percentage of resources utilized from the maximum available for the particular FPGA.

The next most important resource is the distributed memory, particularly for many pipeline stages. It is used for temporary storage and by compute module. By carefully designing the stage to eliminate or reduce duplication in the input buffer and temporary storage, the distributed memory usage is limited to 45% on PYNQ-Z2 and 25% on Ultra96. Our HW accelerator design, it is important that we have only small resource usage so that we can make a more complex circuit with our limited resources.

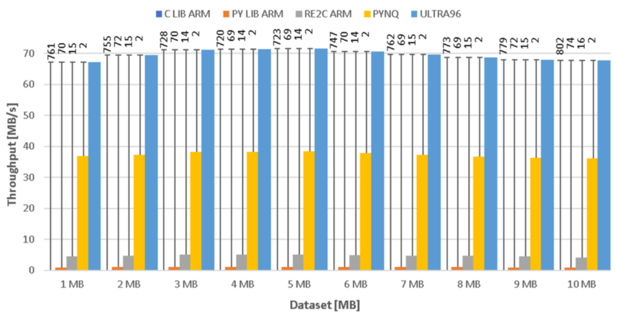


Fig.5: *Speedup of performance in comparison between PYNQ FPGA Accelerator vs. CPU approach.*

Energy Efficiency

We investigated the energy consumption of CPUs on PC, ARM, PYNQ, and FPGA on PYNQ during the evaluations. Using equations 1 and 2, we can calculate and then evaluate the power efficiency.

$$E[J] = P[W] \times t[s] \tag{1}$$

The energy E in joules [J] is equal to the average power P in watts [W], times processing time t in seconds [s]. Power is measured by USB power checker TAP-TST8 for CPU on PC and RT-USBVATM for PYNQ.

$$EE = \frac{Throughput[MB/s]}{W[J/s]} = \frac{ProcessedDatasize[MB]}{Energy[J]} \tag{2}$$

The energy efficiency EE is equal to the throughput [MB/s], divided by watts W in joules per-second [J/s] or Processed data size in Megabytes divided by energy consumption [MB/J].

Table 6: *Energy efficiency evaluation.*

	ULTRA 96	PYNQ-Z2	C LIB CPU	PY LIB CPU	RE2C CPU
E [J]	1.24	1.40	623	24.44	6.84
EE [MB/J]	8	7	0.02	0.12	1.15
P [W]	8.4	5.03	64.9	74	43
t [s]	0.148	0.279	9.596	0.330	0.159
Throughput [MB/s]	68	36	1.06	9.12	49.5

Table 6 shows the energy consumption comparison from evaluations of software and the accelerated cases between CPU, ARM, and the Zynq platforms. The highest power consumption is for Core i7 processors. The DRAMs is 74.0[W] for the Python Library on CPU and 64.9[W] for C Library on CPU. Energy consumption of both Zynq platforms (both the MPSoC FPGA and the DRAM) is 5.03[W] or 1.4 [J] and 8.4[W] or 1.24[J] for PYNQ-Z2 and Ultra96, respectively, when both tested with an RT-USBVATM power meter. Ultra96 energy efficiency for our proposed accelerator is slightly better than with PYNQ-Z2 because of the higher specifications ARM processor, larger DRAM Size, and circuit size. This efficiency is a huge advantage due to lower power consumption and faster execution time.

6. CONCLUSION

In this paper, we presented rules and methods for translating regular expression patterns into supported hardware code. We perform pattern matching and feature extraction from data stream with five different use-cases with seven different implementations for evaluation. We able to get better performance on HW chips compared with existing high-level programming using commonly used libraries (C/C++ and Python regex libraries) and the RE2C toolkit on embedded platforms and processors. Our translation rules for hardware accelerators are proven to allow higher performance than other implementations on optimized software regexes for CPU and ARM processors. With throughput exceeding 88[MB/s], which is up to over 300 times better than ARM evaluation using C Library and up to 74 times that of the Python library. When we compared to CPU usage, both libraries were up to 21 and 39-times as fast, respectively. Next, when compared to RE2C, we achieved up to 16 times on ARM and 1.4 times speedup on CPU. Finally, compared head-to-head with PYNQ-Z2, our proposed accelerators achieved almost double the performance on all case studies and datasets. Furthermore, our accelerator consumed only 1.24[J] energy with an efficiency of 8[MB/J].

Vivado HLS has limitations for C/C++ language expression and support, so we claim that our translation rules are required in the regex system. The environment in which we introduced rules are implemented is operating without a problem, and the design of a regular expression processing circuit has become more accessible. The rules and translation are important for efficient design flow, as it would be impossible to break through the current limitation of HLS. Although it takes a long time to design the FPGA circuit and synthesis it, the circuit data (configuration data) is reusable. Benefiting our translation tool with proposed rules leads to omitting the hand rewriting cost and high productivity.

It is an essential factor for the rapid development of FPGA accelerator design. Furthermore, we demonstrate that it is possible to avoid the high overhead for data communication (data movement) by applying in-memory data processing architectures principles. We develop an architecture that supports streaming data computation by reducing data transfer overheads with in-memory transfer DRAM-DMA (Direct Memory Access) between Processing System (PS) and Programming Logic (PL) on the FPGA accelerator. Resulting in our approach system does not incur significant hardware overheads once compiled. In addition, our rules enable us to solve the complexity of hardware connectivity targeting to improve performance, lower energy consumption, and adapt software algorithms into hardware accelerators.

For future work, we plan to develop a hardware-software co-design framework for rapid prototyping and high productivity by evaluating the different types of FPGAs. It will enable people to develop real-time data processing on the FPGA accelerators for high performance and high energy efficiency. Moreover, considering the small resource usage is from our evaluation, it is possible to make more complex circuits for optimum resource utilization.

References

[1] K. Yasumoto, H. Yamaguchi and H. Shigeno, "Survey of Real-time Processing Technologies of IoT Data Streams," *Journal of Information Processing*, vol. 24, no. 2, pp. 195-202, 2016.

[2] K.S. Bok, D. Kim and J. Yoo, "Complex Event Processing for Sensor Stream Data," *Sensors*, vol. 18, no.9, 2018.

[3] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, O'reilly, 2017.

[4] J. Teubner and L. Woods, *Data Processing on FPGAs*, Springer, 2013.

[5] Vivado Design Suite User Guide High-Level Synthesis UG902. (2019). UG902 (v2020.1) June 3, 2020.

[6] K. Carruthers, "How the internet of things changes everything: The next stage of the digital revolution," *Journal of Telecommunications and the Digital Economy*, vol. 2, no.4, 2014.

[7] R.P. Sidhu and V. Prasanna, "Fast Regular Expression Matching Using FPGAs," *The 9th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 227-238, 2001.

[8] L. Woods, J. Teubner and G. Alonso, "Complex event detection at wire speed with FPGAs," *Proceedings of the VLDB Endowment*, vol .3, pp. 660-669, 2010.

[9] V. Paxson, S. Campbell, C. Leres and J. Lee, *Bro Intrusion Detection System*, 2006.

[10] S. Prithi, S. Sumathi and C. Amuthavalli, *A Survey on Intrusion Detection System using Deep Packet Inspection for Regular Expression Matching*, 2017.

[11] S.A. Shah and B. Issac, "Performance Comparison of Intrusion Detection Systems and Application of Machine Learning to Snort System," *Future Gener. Comput. Syst.*, vol. 80, pp. 157-170, 2018.

[12] L. Woods, J. Teubner and G. Alonso, "Real-time pattern matching with FPGAs," *2011 IEEE 27th International Conference on Data Engineering*, pp. 1292-1295, 2011.

[13] Y. Yang and V. Prasanna, "High-Performance and Compact Architecture for Regular Expression Matching on FPGA," *IEEE Transactions on Computers*, vol. 61, pp. 1013-1025, 2012.

[14] W.P. Kiat, K.M. Mok, W. Lee, H.G. Goh and R. Achar, "An energy efficient FPGA partial reconfiguration based micro-architectural technique for IoT applications," *Microprocess and Microsystems*, vol.73, 2020.

[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol .30, pp. 473-491, 2011.

[16] P. Bumbulis and D. Cowan, "RE2C: a more versatile scanner generator," *ACM Letters on Programming Languages and Systems*, vol. 2, pp. 70-84, 1993.

[17] U. Trofimovich, "RE2C: A lexer generator based on lookahead-TDFA," *Software Impacts*, vol. 6, 2020.

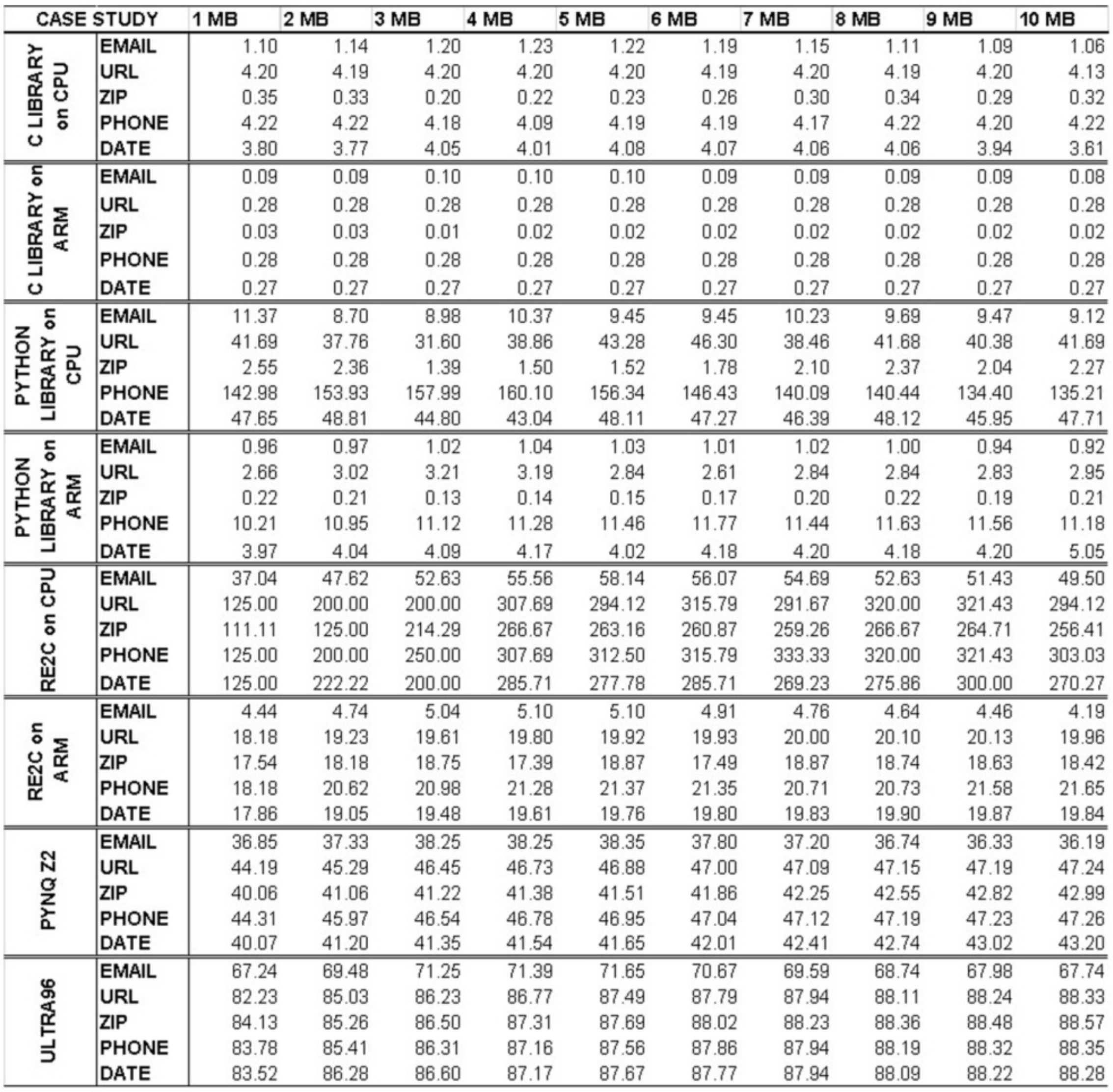
[18] `http://www.pynq.io/`

[19] `https://data.world/brianray/enron-email-dataset`

[20] Hendarmawan, M. Kuga and M. Iida, "Translation Rules of Regular Expression Code for Hardware Accelerator," *Proc. of Asia pacific Conference on Robot IoT System Development and Platform, Digital Library, Information Processing Society of Japan*, pp. 51-58, March 15th, 2021.

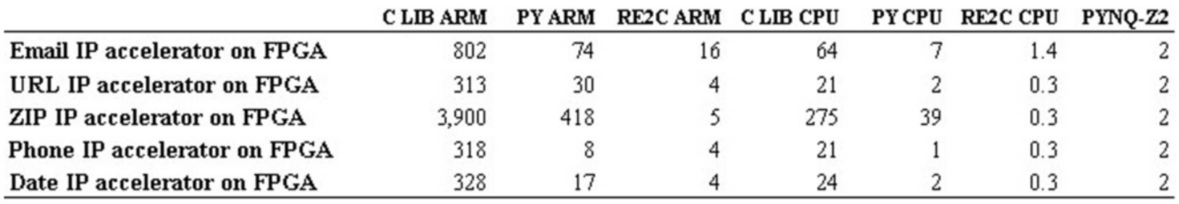
[21] Vivado design suite user guide: High-level synthesis, UG902 (v2019.1) August 12, 2021. [Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf`

APPENDIX 1*. Throughputs for all evaluations [MB/s].*



CASE STUDY		1 MB	2 MB	3 MB	4 MB	5 MB	6 MB	7 MB	8 MB	9 MB	10 MB
C LIBRARY on CPU	EMAIL	1.10	1.14	1.20	1.23	1.22	1.19	1.15	1.11	1.09	1.06
	URL	4.20	4.19	4.20	4.20	4.20	4.19	4.20	4.19	4.20	4.13
	ZIP	0.35	0.33	0.20	0.22	0.23	0.26	0.30	0.34	0.29	0.32
	PHONE	4.22	4.22	4.18	4.09	4.19	4.19	4.17	4.22	4.20	4.22
	DATE	3.80	3.77	4.05	4.01	4.08	4.07	4.06	4.06	3.94	3.61
C LIBRARY on ARM	EMAIL	0.09	0.09	0.10	0.10	0.10	0.09	0.09	0.09	0.09	0.08
	URL	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
	ZIP	0.03	0.03	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.02
	PHONE	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
	DATE	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27
PYTHON LIBRARY on CPU	EMAIL	11.37	8.70	8.98	10.37	9.45	9.45	10.23	9.69	9.47	9.12
	URL	41.69	37.76	31.60	38.86	43.28	46.30	38.46	41.68	40.38	41.69
	ZIP	2.55	2.36	1.39	1.50	1.52	1.78	2.10	2.37	2.04	2.27
	PHONE	142.98	153.93	157.99	160.10	156.34	146.43	140.09	140.44	134.40	135.21
	DATE	47.65	48.81	44.80	43.04	48.11	47.27	46.39	48.12	45.95	47.71
PYTHON LIBRARY on ARM	EMAIL	0.96	0.97	1.02	1.04	1.03	1.01	1.02	1.00	0.94	0.92
	URL	2.66	3.02	3.21	3.19	2.84	2.61	2.84	2.84	2.83	2.95
	ZIP	0.22	0.21	0.13	0.14	0.15	0.17	0.20	0.22	0.19	0.21
	PHONE	10.21	10.95	11.12	11.28	11.46	11.77	11.44	11.63	11.56	11.18
	DATE	3.97	4.04	4.09	4.17	4.02	4.18	4.20	4.18	4.20	5.05
RE2C on CPU	EMAIL	37.04	47.62	52.63	55.56	58.14	56.07	54.69	52.63	51.43	49.50
	URL	125.00	200.00	200.00	307.69	294.12	315.79	291.67	320.00	321.43	294.12
	ZIP	111.11	125.00	214.29	266.67	263.16	260.87	259.26	266.67	264.71	256.41
	PHONE	125.00	200.00	250.00	307.69	312.50	315.79	333.33	320.00	321.43	303.03
	DATE	125.00	222.22	200.00	285.71	277.78	285.71	269.23	275.86	300.00	270.27
RE2C on ARM	EMAIL	4.44	4.74	5.04	5.10	5.10	4.91	4.76	4.64	4.46	4.19
	URL	18.18	19.23	19.61	19.80	19.92	19.93	20.00	20.10	20.13	19.96
	ZIP	17.54	18.18	18.75	17.39	18.87	17.49	18.87	18.74	18.63	18.42
	PHONE	18.18	20.62	20.98	21.28	21.37	21.35	20.71	20.73	21.58	21.65
	DATE	17.86	19.05	19.48	19.61	19.76	19.80	19.83	19.90	19.87	19.84
PYNQ Z2	EMAIL	36.85	37.33	38.25	38.25	38.35	37.80	37.20	36.74	36.33	36.19
	URL	44.19	45.29	46.45	46.73	46.88	47.00	47.09	47.15	47.19	47.24
	ZIP	40.06	41.06	41.22	41.38	41.51	41.86	42.25	42.55	42.82	42.99
	PHONE	44.31	45.97	46.54	46.78	46.95	47.04	47.12	47.19	47.23	47.26
	DATE	40.07	41.20	41.35	41.54	41.65	42.01	42.41	42.74	43.02	43.20
ULTRA96	EMAIL	67.24	69.48	71.25	71.39	71.65	70.67	69.59	68.74	67.98	67.74
	URL	82.23	85.03	86.23	86.77	87.49	87.79	87.94	88.11	88.24	88.33
	ZIP	84.13	85.26	86.50	87.31	87.69	88.02	88.23	88.36	88.48	88.57
	PHONE	83.78	85.41	86.31	87.16	87.56	87.86	87.94	88.19	88.32	88.35
	DATE	83.52	86.28	86.60	87.17	87.67	87.77	87.94	88.09	88.22	88.28

APPENDIX 2*. FPGA Accelerators (Ultra96) vs. other evaluation [♯ speedup].*



	C LIB ARM	PY ARM	RE2C ARM	C LIB CPU	PY CPU	RE2C CPU	PYNQ-Z2
Email IP accelerator on FPGA	802	74	16	64	7	1.4	2
URL IP accelerator on FPGA	313	30	4	21	2	0.3	2
ZIP IP accelerator on FPGA	3,900	418	5	275	39	0.3	2
Phone IP accelerator on FPGA	318	8	4	21	1	0.3	2
Date IP accelerator on FPGA	328	17	4	24	2	0.3	2

Hendarmawan received his BSc in Computer Science from University of Brawijaya, Indonesia, in 2008 and M.Eng in Computer Science from Kumamoto University. He was a junior lecturer at University of Brawijaya. Currently, he is a PhD student at Kumamoto University, Graduate School of Science and Technology. He has been involved in various research projects, including FPGA Accelerator, IoT and Big Data Processing. His research interests include interests include parallel processing, computer architecture, reconfigurable system and design methodology. He is a student member of IEEE.

Morihiro Kuga received his B.Eng. in Electronics from Fukuoka University in 1987 and an M.Eng. and a D.Eng. in Information Systems from Kyushu University in 1989 and 1992. From 1992 to 1998, he was a lecturer at the center for Microelectronic Systems, Kyushu Institute of Technology. He has been an associate professor of computer science at Kumamoto University since 1998. His research interests include parallel processing, computer architecture, reconfigurable system, and VLSI system design. He is a member of the IPSJ and the IEICE.

Masahiro Iida received his B.Eng. in Electronic Engineering from Tokyo Denki University in 1988. He was a research engineer at Mitsubishi Electric Engineering Co., Ltd. from 1988 to 2003. He received his ME in Computer Science from Kyushu Institute of Technology in 1997. He received his DE from Kumamoto University, Japan, in 2002. He was an associate professor at Kumamoto University until 2015, and from 2002 to 2005, he held an additional post as a researcher at PRESTO, Japan Science and Technology Corporation (JST). He has been a professor in the Faculty of Advanced Science and Technology at Kumamoto University since January 2016. His current research interests include high-performance, low-power computer architectures, FPGA computing, VLSI devices, and design methodology. He is a senior member of the IPSJ and the IEICE, and a member of IEEE.