Accelerating string tokenization with FPGAs for IoT data handling equipment

Kazuhiro Yamato *
MIRACLE LINUX CORPORATION

2016/12/1

Abstract

This paper reports on the results of a study to accelerate string tokenization using FPGAs suitable for both IoT gateways and data center servers. The prototype developed with Xilinx High-Level Synthesis software runs at 200 MHz and processes up to 32 ASCII characters per clock cycle. It incorporates either OpenCL or our own framework (Volvox) to transfer data between a host computer and a FPGA board via a PCI Express interface. Evaluations showed processing with a prototype with Volvox was approximately 10 times faster than with a CPU.

1 Introduction

Projections indicate the number of IoT (Internet of Things) devices will exceed 20.8 billion [1] by 2020. A key component in the handling of this enormous array of data and devices will be IoT gateways, which will likely handle certain preprocessing tasks including aggregation, format conversion, and dropping junk data as well as collecting data from devices and forwarding them to the Internet servers. Despite the devices and traffic IoT gateways will be expected to handle, it is generally assumed that they will be installed in locations other than traditional data centers, including locations that impose harsh constraints with respect to space, cooling, and power consumption. Text strings will account for a portion of this traffic, since many network protocols and data formats are based on text data like HTTP and JSON. In addition, if IoT devices begin running lightweight programming languages (e.g., Python and Ruby), which have recently begun running on even small devices, the communication will likely be text based. Hence, the efficient processing is among the key prerequisites for a sophisticated IoT gateway.

Data centers will also face a flood of data from IoT devices, generating numerous opportunities for text processing, including distributed batch processing of collected data (e.g., with Hadoop [2]), syntactic analysis of natural language, AI (Artificial Intelligence) interfaces, and parsing logs from numerous servers.

FPGAs (Field Programmable Gate Arrays) represent an ideal solution to these issues. According to Intel, one-third of cloud service nodes will be equipped with FPGAs by 2020 [3]. FPGAs will allow users and developers to configure circuits tailored to specific purposes. Good FPGA circuit design will improve computing performance and reduce power consumption [4]. However, general software engineers and operators generally find it difficult to use FPGA skillfully, since design requires knowledge of digital circuits as well as the specifics of a host server system, including the system bus and operating systems. In addition, newly developed or modified applications must include additional processing in order to use FPGAs.

We plan to address these challenges by creating an operating system that can use FPGAs transparently by drawing on basic libraries like the GNU C Library (glibc) [5]. The advantage of this approach is that it requires no modification of the application. Additionally, the recent trend toward fusing FPGAs with CPUs raises the possibility this concept will be valid on both Intel and ARM platforms. Intel recently announced the completion of its acquisition of Altera Corporation, one of the leaders in FPGA technology [6]. Xilinx, the other leading company, has already released a SoC (System on Chip) called Zynq that integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA [7]. The CEO of Softbank the parent company of ARM Ltd., has sounded a positive note regarding use of ARM-based processors for IoT [8].

As the first step in tackling this challenge, we created a prototype of a tokenizer that decomposes strings into words, among the most basic tasks in text handling. This paper describes the architecture of the prototype and the results of our evaluations.

*Email: kazuhiro.yamato@miraclelinux.com

2 Prototype

2.1 Overall design

The prototype consists of a host computer and an FPGA board (listed in Table 1) connected to a PCI Express interface (Figure 1). The components in the figure are divided into three layers. The bottom layer (gray background) is hardware with fixed functions. The middle layer (blue background) plays a role in bridging the bottom and the upper layers and handles many functions commonly used by various applications, including DMA transfer and interrupt handling. The upper layer (red background) has application-specific features.

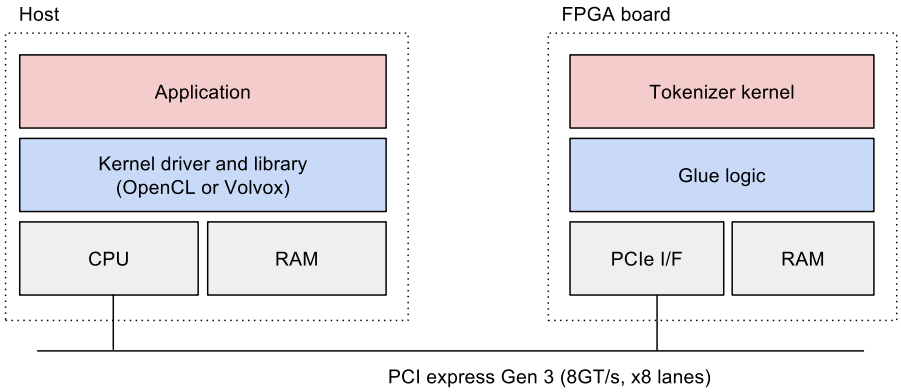

Figure 1: Overall design

Section 2.2 discusses the tokenizer kernel [1], which handles the most essential tasks. For the middle layer, we used two different frameworks, OpenCL and Volvox, shown in 2.3 and 2.4, respectively, and evaluated performance for both cases. Section 2.5 discusses host applications in the upper layer.

Table 1: FPGA board and related specifications

FPGA board	ALPHA DATA ADM-PCIE-7V3 [11]
FPGA device	Xilinx Virtex-7 XC7VX690T-2
Maximum lane width	×8
Maximum link speed	8 GT/s
DDR	Two 8 GB ECC-SODIMM for memory speeds up to 1333 MT/s

2.2 Tokenizer kernel

The tokenizer kernel calculates pairs of a start and an end position for all words in the input strings for each line. Table 2 gives the specifications for the current prototype, which processes up to L (see Table 4) in one run of the kernel. Written in C++ and synthesized in Xilinx Vivado High-Level Synthesis (HLS) 2016.1 [9], the I/O interface is defined like a function of the C language, as listed in Table 3.

Table 2: Specification for tokenizer prototype

Delimiter	Space character
Multibyte character	Not supported
Consecutive delimiters	Treated as one delimiter
Maximum line length	65535

Figure 2 illustrates how to tokenize the two lines '□□`MIRACLE`□□□`LINUX`' and '`Corporate`□`color`□`is`□`green.`' (□ representing a space). The input parameter `num_lines` and `total_length` are, respectively, the number of lines and the summed size of lines processed at one kernel run. `lengths` is an array in which the size of each line is stored. The array size is identical to `num_lines`. The array `lines` contains all the strings to be processed

[1] We call the function configured in an FPGA 'kernel'. Note that this differs from and should not be confused with the Linux kernel.

Table 3: Interface of tokenizer kernel

parameter	I/O	bit width	type	protocol	Elem. in chunk
num_lines	IN	32	scalar	AXI slave (register)	N/A
total_length	IN	32	scalar	AXI slave (register)	N/A
lines	IN	8	array	AXI master	32
lengths	IN	16	array	AXI master	16
markers	OUT	32	array	AXI master	8
positions	OUT	16	array	AXI master	16

Table 4: Configuration parameters of tokenizer kernel

Parameter	Value	Brief description
L	8192	Maximum number of lines the tokenizer can process at once.
W_D	256	Data width of AXI master interface.
N_S	32	Number of Splitters.
Q_p^W	64	Length of burst write for positions.

and has no need for padding or code indicating line breaks. The occurrence of '\n' in a line is handled just like an ordinary character.

The output parameter `markers` is an array with the size of `num_lines`+1. The element points to the head of the region containing tokenization results of the corresponding line. The last element points to the next to the last element of `positions`. An array `positions` contains pairs of a start and an end positions of words. The array size varies depending on input strings and can be calculated as follows with the last element of `markers`:

$$N_p = Z/B_p \tag{1}$$

where N_p, Z, and B_p are the number of elements in `positions`, the last value of `markers`, and the element size of `positions`, respectively. Similarly, the number of tokens N_t is given by

$$N_t = N_p/2 \tag{2}$$

Figure 3 shows a block diagram of the kernel. Since the port for input and output is actually one AXI master interface with a data width of W_D bits, multiple elements in the arrays are read or written at once. We call this unit a *chunk*. The number of elements in a chunk is listed in the column 'Elem. in chunk' in Table 3.

The reader first reads out all elements of `lengths` and then gets `lines`. This approach accesses memory in sequential order and triggers AXI burst reads. Burst reads and writes are data transfer methods for achieving higher throughput by transferring multiple data stored at consecutive addresses in a single operation. Burst transfers are among the most important points for fast data input and output. The dispatcher sends a read chunk of `lines` to one of the splitters cyclically. The splitter parses one character (8 bits) per clock cycle, and its iteration interval is $W_D/8$ clocks [2]. By putting in N_s splitters, which is equal to the iteration interval, we can process in total N_s characters per clock cycle. Since the splitter parses only one of the chunks that forms part of a line, it cannot calculate the complete positions in the line. It also calculates some variables so that complete positions can be calculated in the later block along with them. The linearizer cyclically gathers the parsed results from each splitter and regenerates a single data stream. The unifier calculates complete positions and markers in combination with the results of the previous processing. The advantage of this architecture is that the latency (process time) is independent of the distribution of the input line length and generally proportional to total length.

The calculated positions and markers, respectively, are sent to the positions formatter and the marker formatter. The blocks create chunks whose size is W_D bits from the incoming data elements. Each time Q_p^W of the positions are ready, the writer writes them. On the other hand, markers are buffered in the marker formatter until all positions are written. Since the maximum number of `markers` is $L + 1$, temporary storage poses no problems. This strategy is employed to trigger burst writes of AXI.

We used the AXI master interface with parameters tuned as indicated in Table 5. As described above, the burst lengths of `positions` and `markers` are Q_p^W and $L+1$, respectively. Note that burst length here is a value used in `memcpy()` or a `for` statement. It differs from the AWLEN parameter, a parameter for the AXI interface, and remains at the default value (15). The values in the table were determined heuristically for optimal results. Without the parameters, it took a longer interval between one burst write of `positions` and the next one than would be expected from the Vivado HLS C/RTL co-simulation result.

[2] More than one clock cycle is required to process chunks consisting of more than 16 words.

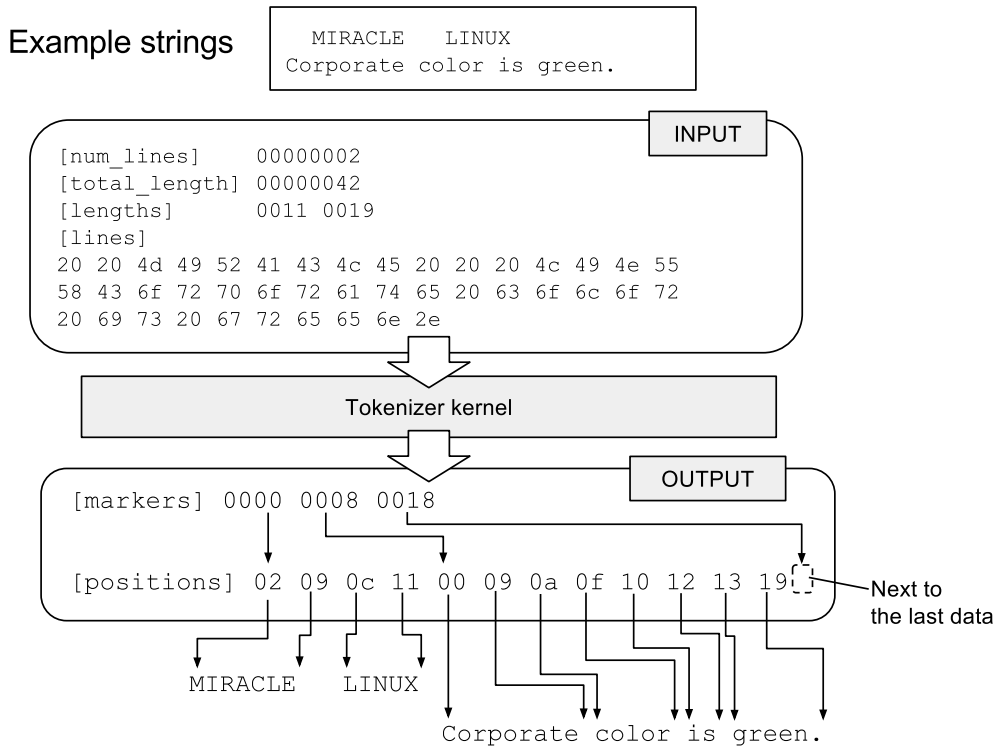

Figure 2: Example of tokenization. For the sake of visibility, note that the values of `markers` and `positions` are shown with shorter bit width than the actual ones.

Table 5: AXI interface parameters specified explicitly

parameter	latency	max_read_burst_length
lines	0	32
lengths	0	32
markers	0	default (16)
positions	0	default (16)

The tokenizer kernel is designed to function at 200 MHz, a fixed requirement, provided OpenCL is used with the ADM-PCIE-7V3 FPGA board. The latency is 179 according to co-simulation if one chunk[3] is used as input and is incremented every N_s byte.

2.3 OpenCL framework

OpenCL, a framework standardized by the Khronos Group [10], allows developers to use various processors, including CPUs, GPUs, FPGAs, and dedicated accelerators with one API set based on the C language. The API set is defined both for a host and a device. In addition, SDAccel, the Xilinx OpenCL development environment, allows FPGA kernels to be written not just in OpenCL C, but in C and C++ with #pragma directives. We used SDAccel and wrote the tokenizer kernel in C++.

Figure 4 shows an example of data flow when using OpenCL. Since OpenCL defines the interfaces (APIs) between the boxes named 'Host application' and 'OpenCL framework on Host', the precise sequence depends on the library implementation. (Note that the purpose of this figure is merely to clarify how the mechanism works.)

The numbers in parentheses at the tops of labels indicate the typical order in which they are used. The labels are placed near the components to which they are related. Although the figure shows data transferred in one direction through RAM on the FPGA board, the actual way in which data is transferred between a host and an FPGA board depends on the specific implementation, as mentioned above. Developers need only to design application-specific items that can be written in OpenCL C (or C/C++), with no need to account for the details of hardware such as PCI Express or DDR RAM. This is among the biggest benefits of using OpenCL.

[3]Size is less than or equal to N_s bytes.

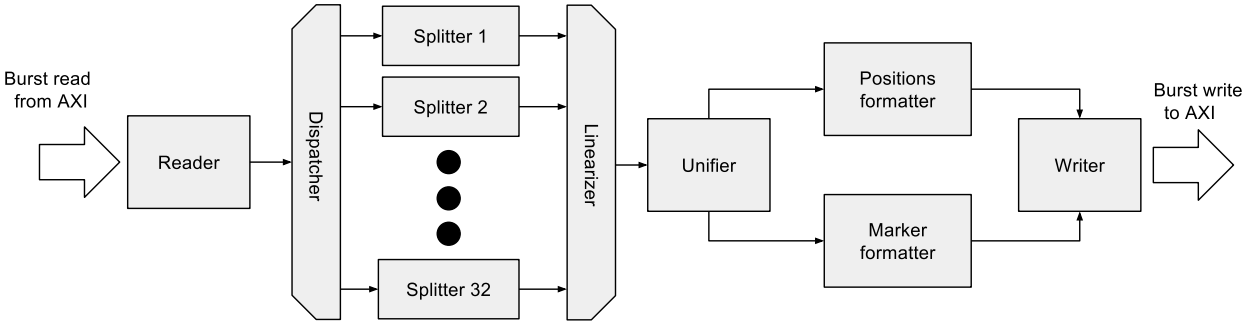

Figure 3: Block diagram of tokenizer kernel

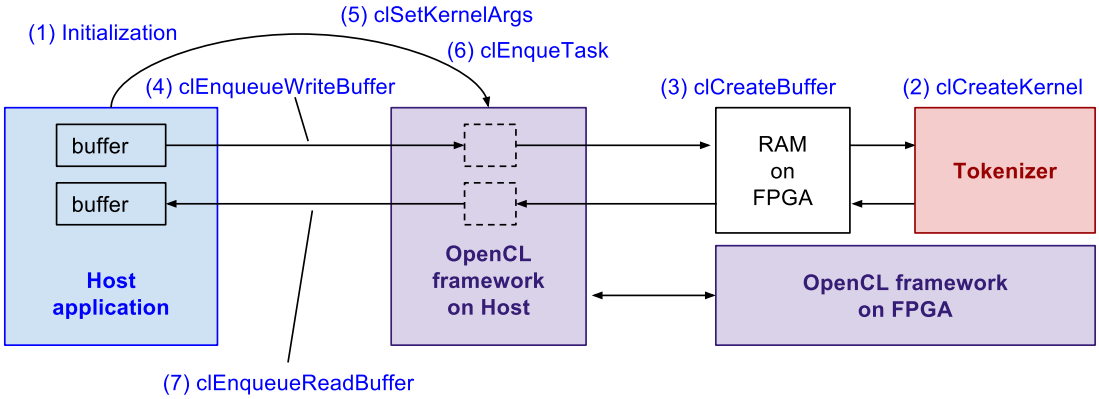

Figure 4: Data flow with Volvox framework

2.4 The Volvox framework

The Volvox framework is software and glue logic for FPGAs that we developed for two reasons: One is that the API set of OpenCL is designed for various purposes and to run on various platforms, meaning it may not be the optimal solution for certain purposes. The other is that SDAccel is not yet in the production status (that is, it is still in beta), and the results may leave room for improvement.

The tokenizer kernel itself can be used for both OpenCL and Volvox frameworks without modification, although the interface to the upper layer in a host is incompatible with that of OpenCL for reasons related to maximum performance. One feature of Volvox is that the tokenizer kernel directly reads and writes data from/to buffers which are mapped in a memory space of an application program, as illustrated in Figure 5. It never uses the DDR RAM on an FPGA board. Additionally, no data is copied between the Linux kernel and an application program.

2.4.1 Glue logic in FPGA

Figure 6 shows the glue logic that enables the tokenizer kernel to access the PCI Express interface with integrated blocks in the FPGA. Except for the tokenizer, all components are Xilinx IPs called LogiCORE, running at 250 MHz with 256 bits of RDATA and WDATA in the AXI interface, which is required to handle PCI Express traffic (×8 lanes, 8GT/s speed).

AXI Bridge for PCI Express Gen3 forwards read and write operations between AXI and PCI Express. That is, a burst read (write) operation of the tokenizer on the AXI interfaces triggers a burst read (write) on PCI Express. This architecture allows the tokenizer to function as a DMA engine for accessing memory on a host.

The interrupt controller is used for the following two purposes.

(1) To convert a level interrupt to an edge interrupt. The tokenizer interrupt pin is automatically synthesized by Vivado HLS; This cannot be modified by the developer.

(2) To generate an interrupt signal synchronized with the clock for AXI Bridge.

Table 6 lists the FPGA resources used after fitting. In terms of usage, the concurrency (number of splitters) in the tokenizer kernel can be increased. However, the logical maximum throughput of the kernel already matches

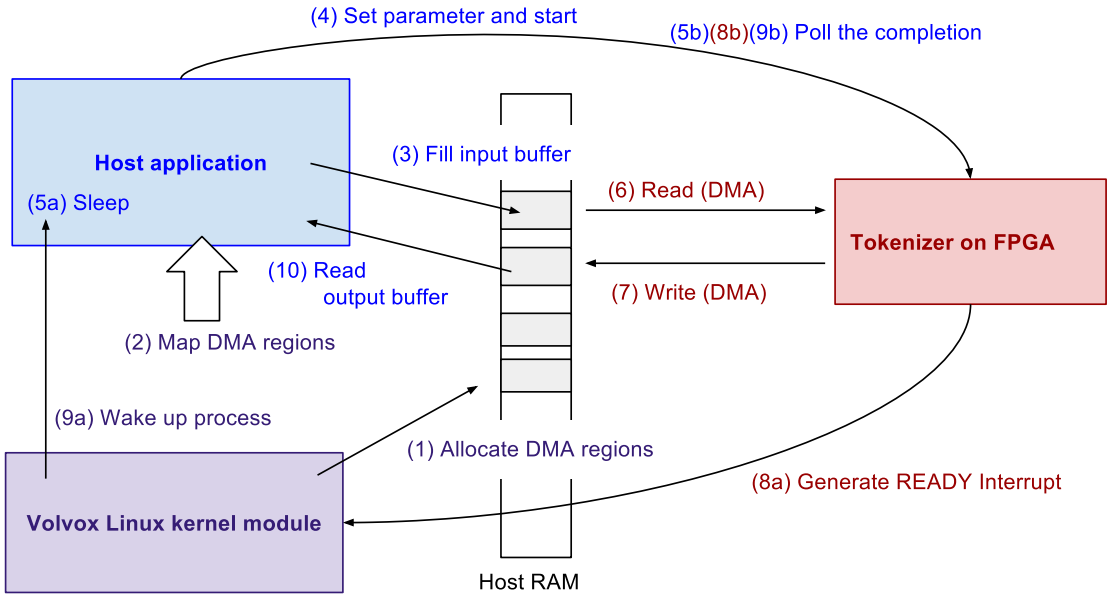

Figure 5: An example of data flow with the Volvox framework

the maximum PCI Express data transfer rate. Therefore, greater concurrency will not improve performance significantly.

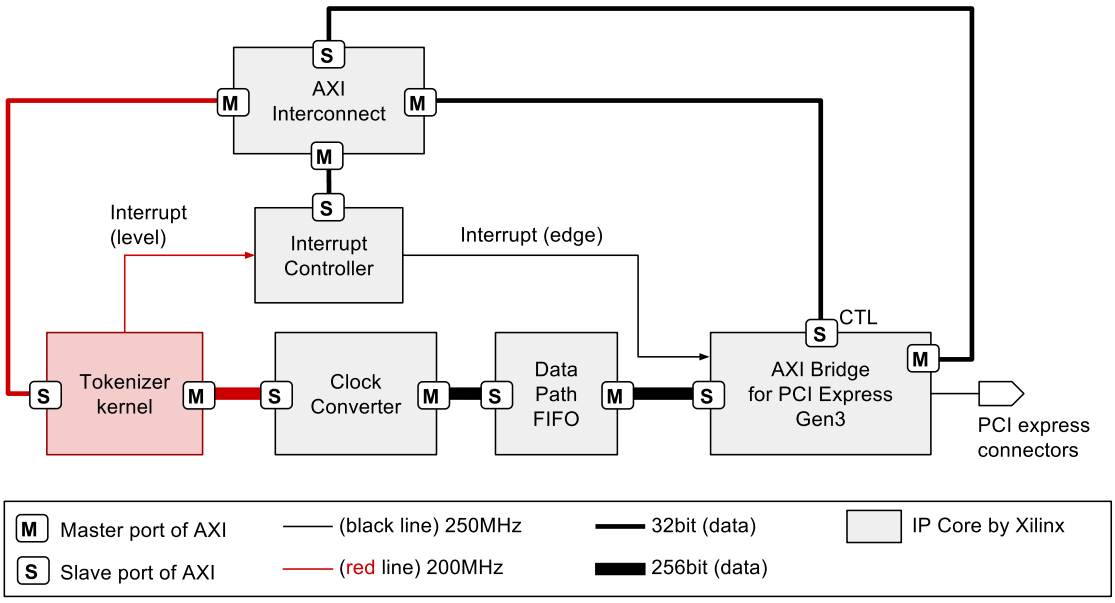

Figure 6: Volvox glue logic

2.4.2 Software on a host computer

The only Volvox framework component on the host side is a loadable Linux kernel module. This allocates two sets of two memory regions, each 4 MiB, for the DMA buffers. One region stores data to the FPGA, while the other stores data from the FPGA. Using two sets of regions enables a host application to prepare input data for the next run or access previously processed data while an FPGA is working with the other set, thereby enhancing performance. The Volvox kernel module maps the regions to a host application via a `mmap` system call.

The kernel module incorporates a function that wakes up a host application on receiving an interrupt from the tokenizer. Nevertheless, using the interrupt remains an option. Since the tokenizer kernel provides a

Table 6: Resources for tokenizer and Volvox glue logic

Component	LUT	LUTRAM	FF	BRAM
tokenizer and Volvox	20%	7%	11%	4%

register, including bits to indicate processing status, a host application can poll the register to determine if a run is complete. The kernel module also incorporates a feature for synchronizing DMA buffers based on the streaming Linux DMA API [12]. A host application must flush and invalidate cache memory using this function.

2.5 Host applications for evaluation

Table 7 summarizes the six host applications developed for performance evaluations, divided into two groups, described, respectively, in Section 2.5.1 and 2.5.2.

2.5.1 Host applications for actual performance measurements

The purpose of the first group is to measure actual performance. Application FO_a and FV_a (a) read an input file; (b) format content for use by the tokenizer kernel; (c) write to the buffer; and (d) run the kernel in the FPGA. When the number of lines of the file exceeds L, steps (c) and (d) above are repeated. Application C_a was developed to determine the time required to do the same thing with a CPU. In contrast to FO_a and FV_a, it does tokenization line by line for simple implementation.

These applications measure the time from the end of the preparation of the first input data to the completion of the run of the tokenizer kernel. To reduce the latency of reading an input file, all strings are once copied at the same time to the buffer in the application in advance. (Measurement excludes this time.)

2.5.2 Host applications for proof of concept

The purpose of another group is to inspect performance with a function similar to `strtok()`, a functions from the C library (glibc). Here, we name the similar function `strtok_v()`. The applications FO_s and FV_s (C_s) count how many times `strtok_v()` (`strtok()`) are called during a set time. As described in Section 1, our goal is to create an OS that uses FPGA transparently. This is one demonstration of the concept. Note that the current implementation is not completely transparent to allow use without any modification of the caller applications.

The tokenizer kernel is assumed to process multiple lines at once. While it can handle a single line, doing so would be inefficient due to overhead, as discussed in Section 3.4.1. It must access multiple lines to be tokenized and run tokenizer with them before they are passed to `strtok_v()`. Nonetheless, even with this limitation, it can (for example) be applied to the batch processing of a flood of logs.

Applications FO_s and FV_s first obtain positions and markers of multiple lines via of FO_a and FV_a, respectively. They then fill a NULL terminator into the given string as a parameter for every call of `char *strtok_v(char *str)`. The argument `str` is a string to be tokenized just like the original `strtok`. The return pointer also has the same meaning as the original. The difference from the original `strtok` is that the delimiter is fixed and cannot be specified. Application C_s simply calls `strtok()` with every line. While implementation of the tokenization engine of C_a is our own, this is a part of the GNU C Library developed by third parties and available to all.

Table 7: Host applications

Application	FO_a	FV_a	C_a	FO_s	FV_s	C_s
Purpose	actual performance			strtok-like function		
Device	FPGA	FPGA	CPU	FPGA	FPGA	CPU
Framework	OpenCL	Volvox	N/A	OpenCL	Volvox	N/A

3 Evaluation

3.1 Host machine

Table 8 gives the specifications for the host machine used in the evaluation.

Table 8: Host machine specifications

CPU	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
RAM	8 GB ×4
OS	CentOS 6.8 (x86_64)
Maximum payload size on PCIe	128

3.2 Input data

For input data, we used nine sizes from 10 to 10^9 bytes of input data consisting of English words. Table 9 lists the number of words and lines for the various inputs. Figures 7 and 8 show the distribution of word lengths and lengths of consecutive separators. The most common word length is 3; 90% and 99% of the words are smaller than 9 and 13 characters, respectively.

The input data size depends on the combination of input words and naturally differs from each run, even if the number of lines is the same. When the number of lines is at the maximum (i.e., L), the data size is roughly several hundred kilobytes.

Table 9: Numbers of words and lines in input data

size (Bytes)	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
words	2	18	154	1579	17194	171422	1743974	17237031	172432363
lines	1	3	43	247	1831	17390	194936	1988957	19911073

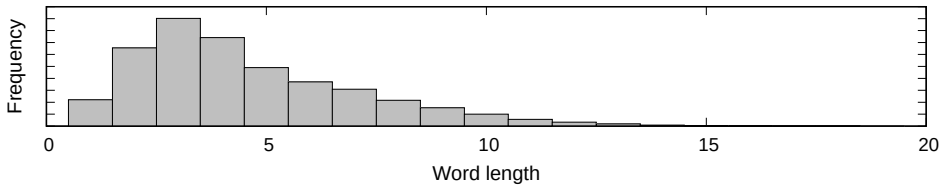

Figure 7: Distribution of word lengths in input data.

3.3 Applications

We used the applications described in 2.5 in polling mode to achieve the maximum performance. Using the interrupt added a latency of approximately 5 to 10 μs to each run of the tokenizer kernel.

The process time is measured with `clock_gettime(CLOCK_MONOTONIC)` placed in the application. The function is called twice per measurement. The latency of the function itself is roughly 50 nanoseconds according to this report [13] and can be regarded as negligible.

3.4 Results and Discussion

3.4.1 Actual performance

Figure 9 shows the process time T for input data size A. Black, blue, and red markers represent measured results with the applications FO_a, C_a, and FV_a. The time for C_a (T_{C_a}) is the shortest where $A \lesssim 10^3$; the time for FV_a (T_{FV_a}) is smaller than that of C_a where $A \gtrsim 10^4$. The ratio T_{C_a}/T_{FV_a} and the throughput of FV_a, the data size processed per second, are approximately 10 and 4.13×10^9 B/s, respectively, where $A \gtrsim 10^6$.

The process time for C_a is roughly proportional[4] to the input data size, while that of FV_a is almost constant and approximately 7 μs where $A \lesssim 10^3$. We examined a breakdown of the time at $A = 10$. FV_a has three subprocesses, as shown as below.

(1) Writing input data to DMA buffer and setup registers of tokenizer

(2) Synchronization of DMA buffer

(3) Run of tokenizer kernel

[4]Nonlinearity around small input data size is considered due to the latency of `clock_gettime()`, as mentioned in 3.3.

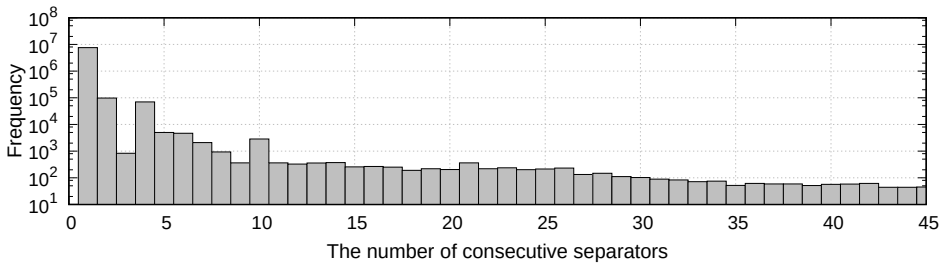

Figure 8: The distribution of the length of consecutive separators in the input data.

The times were approximately 1, 4, and 2 μs, respectively. While the time for subprocesses (1) and (3) increased with input size, that of (2) remained virtually fixed.

The time for (3) implies that the actual minimum latency of a tokenizer is about 400, since the clock frequency is 200 MHz, although the design (C/RTL co-simulation) value is 179, as described in Section 2.2. The difference is due to the latency associated with fetching data from host memory. Figure 10 shows some signals of the AXI interface of a tokenizer obtained with ILA (Integrated Logic Analyzer) [14], a logic analyzer for the actual signals inside an FPGA. The interval between the first rise of ARVALID (start of the read request) and the corresponding rise of RVALID (arrival of the requested data) is about 110 clock cycles. Because the tokenizer performs two burst reads for `lengths` and `lines`, a double latency is added to the value expected from design. Using a single array containing both lengths and lines will reduce latency, if to the detriment of readability and maintainability.

The results for FO_a feature wide variations in process times around $10^-3 \lesssim T \lesssim 10^0$ where $A < 10^7$ and the average time is 10^4 and 10^5 times larger than that of FV_a and C_a where $A = 10$. The reason for this is unknown. The root cause is hard to identify because the OpenCL implementation is proprietary and closed. However, these can be ignored when processing large strings ($A > 10^8$).

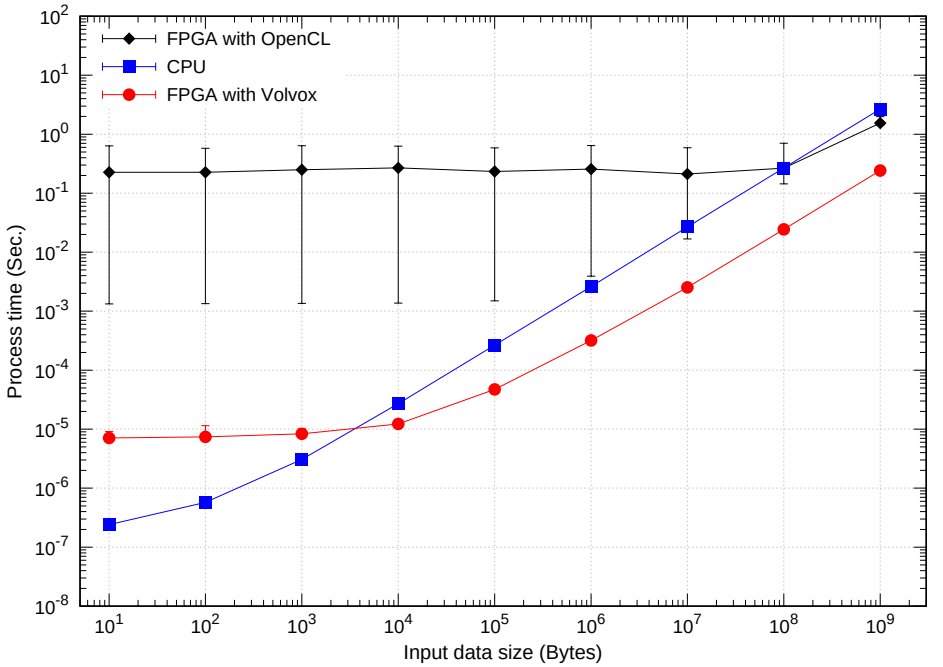

Figure 9: Process time vs. input data size. The markers show the average time. The error bars represent the interval that includes 90 % of the results.

3.4.2 Call count of strtok-like functions

Figure 11 shows the measured call count in 1 second. The overall result is consistent with that of actual performance. The counts ratio of FV_s to C_s is approximately 7, which is smaller than 10, the ratio of the actual performance (T_{C_a}/T_{FV_a}) discussed in the previous section. This is believed to the increase in CPU processing for calling `strtok_v()` and filling NULL terminators.

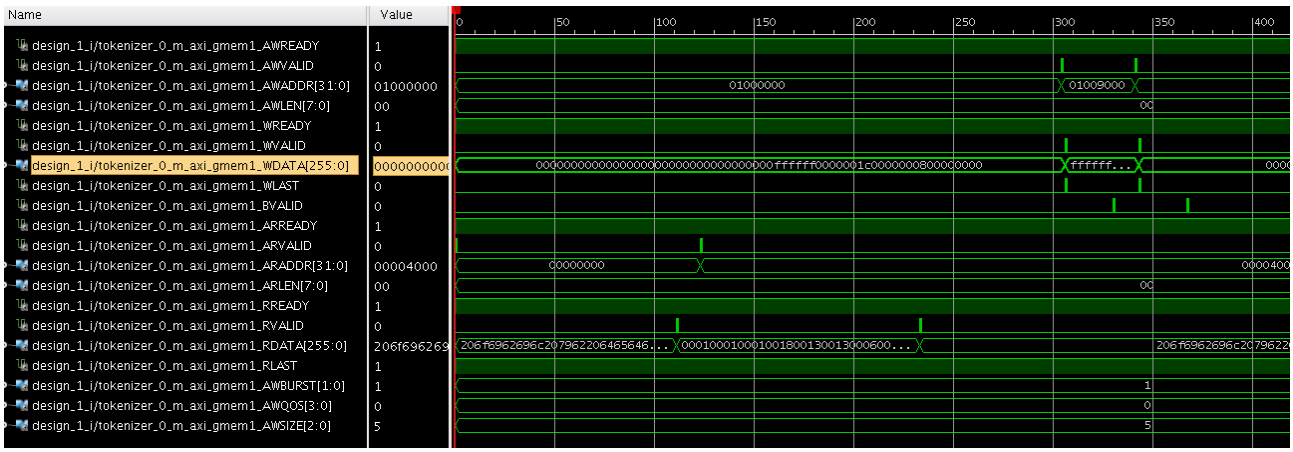

Figure 10: AXI interface signals of a tokenizer when 10 bytes of a string are tokenized.

In general, more processing is required for tokenized words that involve conversion to number type, capitalization, or creation of new strings with the words. Since these can be done by a CPU while the tokenizer kernel is handling processing for the next lines, it would appear that processing of tokenization imposes vanishingly small CPU loads. In any case, the results show the concept is viable and valid.

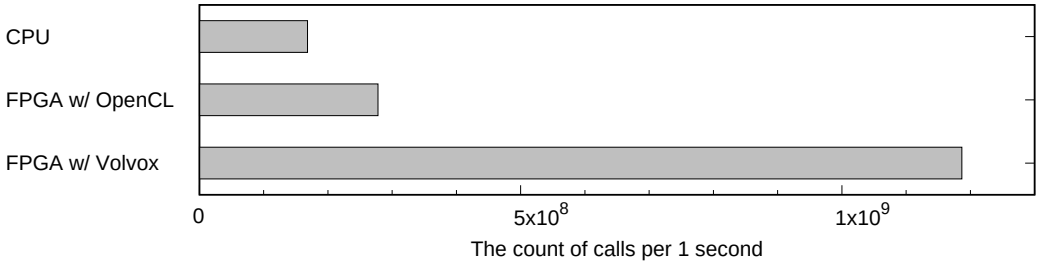

Figure 11: Comparison of call count of `strtok()` and the corresponding function `strtok_v()`.

4 Summary

This paper reports on a study involving tokenizing strings with FPGAs for IoT equipment, whose numbers are expected to grow dramatically. The prototype kernel developed using a Xilinx High-Level synthesis compiler runs at 200 MHz and processes up to 32 ASCII characters per clock cycle. The results showed that throughput with a prototype created with Volvox (a framework we developed for our own purposes) and carefully tuned parameters tuned especially for burst transfers was 10 times faster that with a CPU. Our goal is to create an OS that utilizes FPGAs as described in Section 1. Hence, we plan to implement other functions for text processing and run the combined functions on an FPGA.

We are grateful to Xilinx Corp. for permitting to use the SDAccel, OpenCL development environment, which is currently at the beta stage. While our experience with OpenCL included some unaccountable results, we intend to do more work to help clarify these issues.

References

[1] Gartner's press release, Gartner says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015.

[2] http://hadoop.apache.org/

[3] http://simplecore.intel.com/newsroom/wp-content/uploads/sites/11/2016/03/Intel-Investor-Conference-Call-Deck.pdf

[4] https://www.xilinx.com/products/technology/power.html

[5] https://www.gnu.org/software/libc/

[6] https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[7] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[8] http://www.armtechcon.com/from-trilobites-to-a-trillion-chips-the-iot-explosion/

[9] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[10] https://www.khronos.org/

[11] http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3

[12] https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

[13] http://btorpey.github.io/blog/2014/02/18/clock-sources-in-linux/

[14] https://www.xilinx.com/products/intellectual-property/ila.html