Low-Latency, Line-Rate Variable-Length Field Parsing for 100+ Gb/s Ethernet

Greg Stitt
gstitt@ufl.edu
University of Florida
Gainesville, FL, USA

Wesley Piard
wespiard@ufl.edu
University of Florida
Gainesville, FL, USA

Christopher Crary
ccrary@ufl.edu
University of Florida
Gainesville, FL, USA

ABSTRACT

Field-programmable gate arrays (FPGAs) are widely employed in network-interface cards across applications including cloud services, machine learning, and high-frequency trading. These applications often share a common optimization goal: minimizing latency while meeting throughput constraints. In addition, these applications ideally aim to achieve "line-rate" operation, where the FPGA operates at full bandwidth without using back-pressure to stall incoming data. However, these goals are often conflicting. For example, to minimize latency, application protocols must effectively utilize network bandwidth by encoding variable-length data in variable-length fields. However, variable-length fields often have prohibitively complex processing requirements that prevent line-rate throughput or have excessive latency. In this paper, we present a novel variable-length field parser capable of scaling to accommodate the bus widths and clock frequencies necessary for 100+ Gb/s Ethernet, while still achieving low latency. Our experiments demonstrate parsing variable-length fields at line rate for anticipated bus widths and throughputs, achieving ultra-low latencies under 2 ns for some use cases. To the best of our knowledge, this latency surpasses existing work, including fixed-length field parsing.

CCS CONCEPTS

• **Hardware → Hardware accelerators**.

KEYWORDS

FPGA, SmartNIC, low-latency, HFT

ACM Reference Format:
Greg Stitt, Wesley Piard, and Christopher Crary. 2024. Low-Latency, Line-Rate Variable-Length Field Parsing for 100+ Gb/s Ethernet. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3626202.3637559

1 INTRODUCTION

Previous studies have demonstrated the effectiveness of FPGAs as accelerators across a wide array of domains [2, 11, 13, 14, 19]. FPGA advantages are particularly pronounced when integrated into network-interface cards, often denoted as SmartNICs [7, 9, 10, 12, 18]. This integration facilitates low-latency processing directly within the network, eliminating the need for data exchange with a host microprocessor. The combination of low-latency, high-throughput, and low-energy acceleration has made SmartNIC FPGAs an attractive technology, especially for cloud services, machine learning, networking, and high-frequency trading.

SmartNIC FPGAs, positioned in close proximity to networks, are routinely tasked with parsing and processing network traffic. While the accelerated parsing of Ethernet, IP, and transport-layer protocols (such as TCP and UDP) represents a mature technology, the parsing of application protocols remains an evolving field due to changing requirements. Transport-layer (and lower) protocols predominantly utilize fixed-length fields. Though payloads may exhibit variable lengths, parsed headers consistently contain fields within the same bit range for each new packet.

By contrast, application-layer protocols often entail variable-length data, such as strings and ticker symbols. Ideally, these variable-length data should be encoded within variable-length fields to maximize network bandwidth utilization. However, hardware acceleration of variable-length fields, especially in the context of wide buses, poses considerable challenges. Firstly, the start of each field can occur at any bit position within a given bus input. Secondly, different inputs from the bus can contain different numbers of fields. Finally, fields may overlap multiple consecutive bus inputs. Consequently, the FPGA must dynamically align all variable-length fields within a bus while preserving the contents of fields spanning multiple inputs.

Addressing these challenges in general, particularly in the context of achieving line rate, demands a hardware solution capable of aligning and coalescing an arbitrary number of variable-length strings *every cycle*—a task of significant complexity. Additionally, many use cases pose further challenges by prioritizing minimization of latency. In essence, the parsing of variable-length fields necessitates solutions that reconcile competing objectives: high throughput via substantial parallelism and high clock frequencies, while simultaneously achieving low latency.

In this paper, we present an FPGA architecture capable of efficiently processing any number of variable-length fields within a single bus input, while scaling to accommodate wide bus sizes and high frequencies required for 100+ Gb/s Ethernet. The architecture offers numerous configurable options for optimizing trade-offs between throughput and latency for different bus widths. Our experiments thoroughly investigate these trade-offs, demonstrating latencies under 2 ns at 10 Gb/s, 2.7 ns at 40 Gb/s, and 9.9 ns at 100 Gb/s for some use cases. To our knowledge, these trade-offs

FPGA '24, March 3–5, 2024, Monterey, CA, USA

ACM ISBN 979-8-4007-0418-5/24/03…$15.00
https://doi.org/10.1145/3626202.3637559

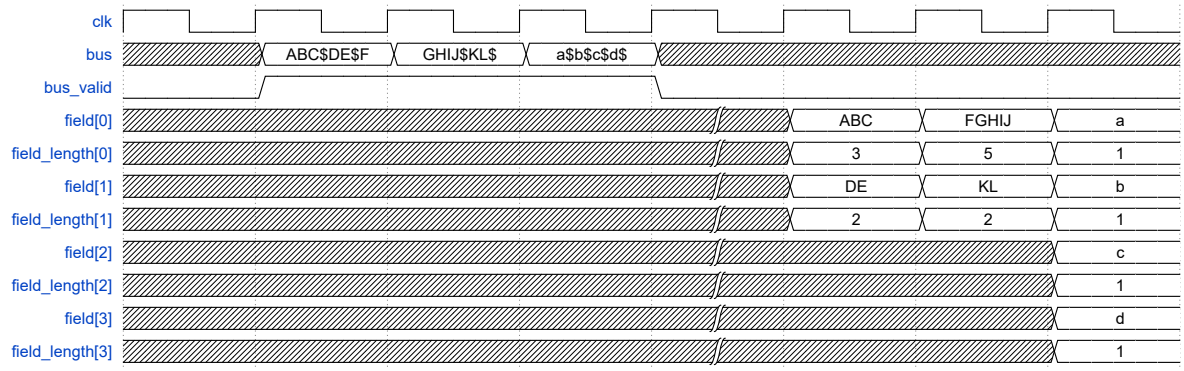


Figure 1: Timing diagram for example with 8-byte bus and $ delimiter. There are four output fields to match the maximum possible number of fields in one input. Note that string FGHIJ spans multiple inputs.

represent a significant improvement over the state-of-the-art [17], even compared to fixed-length field parsing [3].

2 RELATED WORK

Although GPUs are the most common accelerator in general, to our knowledge, GPUs have not yet been integrated with SmartNICs. GPUs are used for network processing by offloading network traffic from a NIC to a PCIe GPU [15], but such an approach is not appropriate for low-latency use cases due to latencies over 80 microseconds [20]. This paper focuses on low-latency, line-rate processing that achieves latencies that are 1000× to 10000× smaller than GPUs.

Numerous FPGA studies have investigated packet processing using SmartNICs for network functions (e.g., [4, 5, 8]). hXDP [6] implemented a software-programmable packet processor on an FPGA-based SmartNIC capable of any function. hXDP was intended to maximize flexibility at the cost of latency and throughput, which was on the order of microseconds and tens of Mb/s. FlowBlaze [16] is conceptually similar, providing a flexible abstraction for building stateful packet processing in FPGAs. FlowBlaze achieved similar performance as hXDP, with latencies in the microseconds, but with a higher throughputs of 40 Gb/s. By contrast, our architecture is specifically for variable-length field parsing and achieves nanosecond latencies for throughputs between 10 and 100 Gb/s.

Similarly, FPGA-based SmartNICs have also been used heavily for low-latency cloud and edge services, including homomorphic computing [18], web searches [7, 10], and machine learning [9, 12], among others. These studies are complementary to our work, and they could potentially leverage our work to minimize latency or increase throughput for variable-length fields.

Sierra et al. [17] presented conceptually similar work focusing on the problem of decoding variable-length data for unstructured meshes streamed from memory. Sierra's work did not optimize for low latency and line-rate processing, and instead used flow control to stall the input stream when the FPGA could not keep up with the data rate. By contrast, our architecture includes numerous configuration options to achieve a wide range of trade-offs between latency and throughput, while guaranteeing line-rate processing.

3 PROBLEM DEFINITION

This paper focuses on the problem of parsing variable-length fields in application-level protocols at line rate. Figure 1 illustrates a timing diagram demonstrating the desired functionality. In this example, the field primitives are 1-byte characters, but primitives could be any bit width in general.

As input, a bus delivers a fixed amount of data, potentially each cycle. We denote the bit width of the bus as w, and use b as the width in bytes ($b = w/8$). The b bytes comprise both complete and partial variable-length fields that are separated by a delimiter, shown as $ in the figure. The architecture allows for configuration of the number of field outputs m, which can vary depending on the specific protocol. In the worst case for $b = 8$, every field is two bytes: a single character followed by a delimiter. Consequentially, the maximum possible number of fields per input is $b/2 = 4$. To support the worst case, m would also be equivalent to $b/2$, but m can also be restricted to smaller values based on characteristics of the specific use case.

In this example, the initial valid data on the *bus* (indicated by the assertion of *bus_valid*) contains two complete fields (ABC and DE), along with one partial field (F), which will be completed by a future input. To minimize post-processing latency, the architecture always outputs fields as soon as possible, rather than waiting for a specific number of fields. This policy is shown in the initial outputs from the architecture, where the first two outputs deliver the fields ABC and DE, in addition to their size. The architecture also provides valid signals for each output field, which are omitted here for brevity.

During the second input cycle, the bus provides a continuation (GHIJ) of the previously incomplete field (F) from the preceding input, as well as another complete field (KL). The architecture outputs those two fields one cycle after the initial output.

It is worth noting that in the first two output cycles, output fields 2 and 3 remained unused because there were not sufficient input fields to necessitate their utilization. However, during the third input cycle, the bus delivers four 1-character fields, resulting in all four outputs containing valid data.

While some application protocols consist entirely of variable-length fields (e.g., Financial Information eXchange (FIX) [1]), most

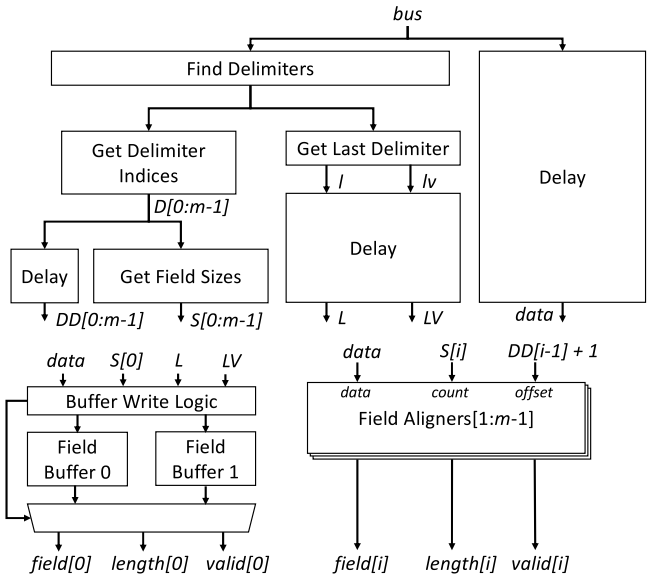


Figure 2: Overview of the variable-length field parsing architecture.

protocols combine both variable and fixed-length fields. The presented architecture remains applicable to such protocols with straightforward adjustments to exclude delimiters within the range of the fixed-length fields and to accommodate the fixed-length shift required by those fields. Our decision to focus exclusively on variable-length fields was intended to isolate and thoroughly explore solutions to this specific problem, rather than creating protocol-specific solutions, which would likely further improve performance via protocol-specific optimizations.

4 VARIABLE-LENGTH FIELD PARSER

4.1 Overview

Figure 2 provides a high-level overview of the variable-length field parsing architecture. We designed every component in this architecture with configurable pipelining options ranging from no pipelining to including a register after every logic level in each targeted FPGA. The inclusion of these configuration options allowed us to explore trade-offs between latency, clock frequency, throughput, and resource utilization to better support different use cases. We also included configuration options to optimize various components (e.g., muxes, priority encoders) to best align with the logic resources of each targeted FPGA.

In the absence of pipelining, the delay components function as wires, and the only registers in the entire architecture are within the field buffers, serving to store partial fields. This configuration allows for the immediate appearance of all complete fields on the outputs within the *same* cycle in which the input is valid. Conversely, with full pipelining, every level of logic has a register, leading to high clock frequencies at the cost of increased latency.

In addition to configuration options for pipelining, the architecture has parameters for specifying the bus' byte width b, the maximum number of fields per input m, the maximum number of bytes per field n, and the delimiter character. For all configuration options, the architecture is capable of taking new bus inputs every cycle (i.e., line rate).

Initially the bus delivers data to the architecture, which goes into the *Find Delimiters* unit, in addition to being delayed to align with subsequent logic. *Find Delimiters* uses b separate 8-bit comparators to compare every byte of the bus with the delimiter character. The output of the *Find Delimiters* unit is b bits, where each bit is asserted if the delimiter exists at the corresponding byte of the input.

To output each field, the architecture must first shift the fields such that the first byte of the field aligns with the first byte of the corresponding output. While sub-line-rate execution has many options for such alignment, line-rate execution requires a barrel shifter, at least for low latencies. The barrel shifters are implemented in the *field buffers* and *field aligners*, but first the architecture needs to determine how much to shift the bus data for each field. To determine the shift amounts, the *Get Delimiter Indices* unit is used to convert the byte locations of the delimiters into binary-encoded indices that specify the ending of each field. The *Get Delimiter Indices* unit is implemented using a multiple-output priority encoder, which we discuss in Section 4.2.

Get Delimiter Indices outputs up to m indices $D[0 : m-1]$, which the architecture then uses in the *Get Field Sizes* unit to determine the number of bytes in each field with the following equations, where $S[i]$ is the number of bytes of field i provided by the current input, and $|D|$ is the number of delimiters:

$$S[0] = \begin{cases} b & \text{if } |D| = 0 \\ D[0] & \text{if } |D| > 0 \end{cases}$$

$$S[i] = D[i] - D[i-1] - 1, \text{ for } 0 < i < |D|$$

The first field in the input has a special condition because there may exist bus inputs with no delimiter ($|D| = 0$), in which case the number of bytes is just the size of the bus. For all other situations, the bytes in the first field provided by the current input is just the index of the first delimiter $D[0]$. For subsequent fields, the number of bytes is the distance between adjacent pairs of delimiters minus 1. For example, in Figure 1, the DE field in the first input cycle ends at index 6. The preceding field ends at index 3, making the size 2 ($6 - 3 - 1 = 2$). Like all other logic, *Get Field Sizes* has configuration options to register the subtractions.

The delimiter indices (D) are also delayed by the latency of *Get Field Sizes* to ensure they (DD) can be provided to the *Field Aligners* at the same time as the fields sizes (S).

Although *Get Delimiter Indices* specifies the location of all delimiters, identifying the last delimiter is inefficient due to the variable amount of delimiters. This identification at line rate would require a mux that selects from all possible indices, where the select logic requires another priority encoder to identify the last asserted bit. To avoid this complexity, the architecture uses *Get Last Delimiter*, which is a single-output pipelined priority encoder, discussed in Section 4.2. *Get Last Delimiter* first reverses the order of the bits provided by *Find Delimiters* to ensure that it reports the last delimiter. The architecture delays the last delimiter l and its valid signal lv to align with the output of *Get Field Sizes*.

The remaining logic is responsible for assembling the output fields. To handle fields that are partially provided by a single input,

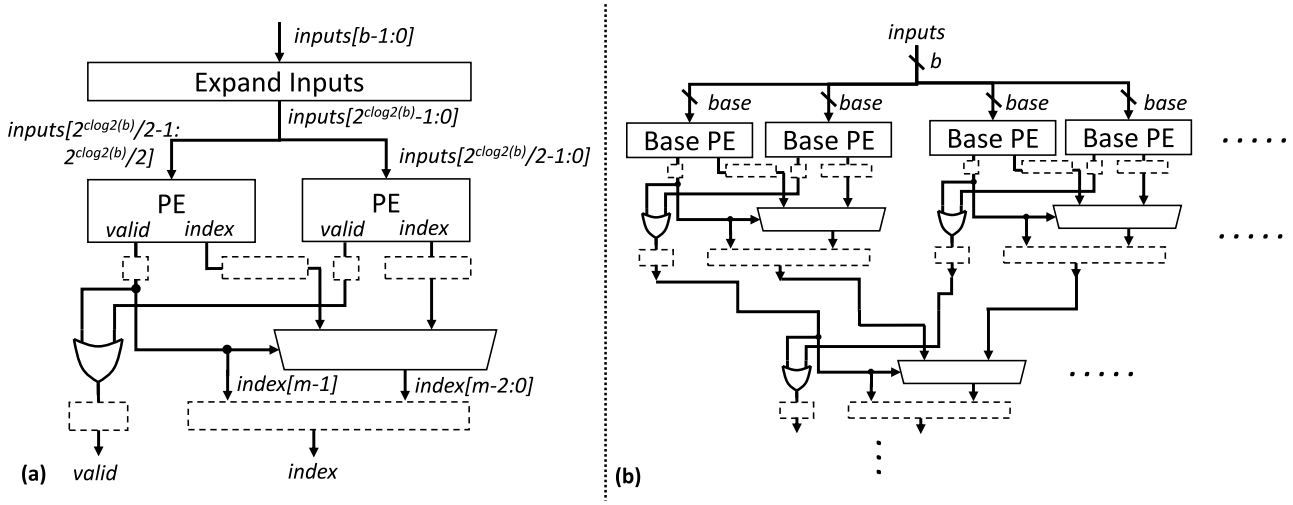


Figure 3: Overview of the pipelined priority encoder, which leverages (a) a recursive definition that divides a priority encoder into two smaller priority encoders, followed by merging logic. (b) This recursive rule can be applied any number of times to create a priority encoder with a single level of logic in between registers by choosing the number of base inputs based on the targeted FPGA. All dashed rectangles are optional pipeline stages.

the *Field Buffers* (Section 4.3) and *Buffer Write Logic* store partial data until a complete field is available.

For all fields that are completely specified by a single bus input (e.g., DE, KL in Figure 1), the *Field Aligners* (Section 4.4) shift the delayed bus data by the appropriate amount to align the corresponding field with the first byte of the field output, while also outputting the length of the field.

For any maximum number of fields per input m, the architecture always has two field buffers. This exact amount is required because for every input provided by the bus, there can be two sets of partial data: one at the beginning of the input that completes a previously buffered field (referred to as the *first field*), and another at the end that starts a new partial field (i.e., the last field).

One challenge of the write buffer logic is that the *field[0]* output must always represent the first completed field within an input, but a field that starts at the end of one input is always completed at the beginning of a subsequent input. As a result, the architecture can't statically designate one field buffer for the first field, and another for the last field. To handle this situation, the architecture resolves the potential ambiguity about which field buffer stores the next field by maintaining a pointer to one of the two field buffers. That pointer then provides a select to the mux to output the correct field. The buffer write logic toggles the field buffer pointer any time there is a valid delimiter anywhere in the input. The presence of a delimiter suggests that any field buffered from a previous input is now complete, so the last field of the current input should now become the first field of the next output.

4.2 Pipelined, Multi-output Priority Encoder

The *Get Delimiter Indices* unit from Figure 2 takes in a vector of bits that specify valid delimiter locations in the bus input. The output of *Get Delimiter Indices* is a set of up to m binary-encoded byte indices for the delimiters.

This functionality is equivalent to a multi-output priority encoder, where instead of outputting the index of the highest-priority asserted input, it instead outputs m indices for the highest-priority m inputs, in sorted order.

Before discussing how to create a pipelined multi-output priority encoder, we first need a way of pipelining a traditional priority encoder so that, if requested, we can register the design to ensure that the logic delay does not increase with wider bus widths. This goal is challenging because traditional priority encoders have a delay that increases logarithmically with the number of inputs, with the base of the logarithm being the number of lookup-table (LUT) inputs in the targeted FPGA.

Figure 3(a) explains our pipelined priority encoder, which leverages the illustrated recursive rule to divide a priority encoder with many inputs into multiple, optionally pipelined levels with a maximum number of inputs per level. The priority encoder takes as input a parameter to specify the base number of inputs before we divide the priority encoder into multiple levels. This parameter allows the user to optimize the priority encoder to the LUTs provided by a given FPGA.

When the actual number of inputs is less than the base inputs, the architecture simply allocates a normal priority encoder. When the number of inputs exceeds the base inputs, the architecture recursively sub-divides the priority encoder into two sub-encoders, where the input is first expanded to the next power of 2 if it isn't already a power of 2.

The two sub-encoders output an index and valid bit for their corresponding half of the expanded input. The rest of the recursive rule then combines these by checking the valid bit of the upper encoder. If the upper valid bit is asserted, the architecture asserts the most-significant bit of the output index, and sets the other index bits equal to the index of the upper encoder. Otherwise, the most-significant bit of the output index is cleared and the remaining index

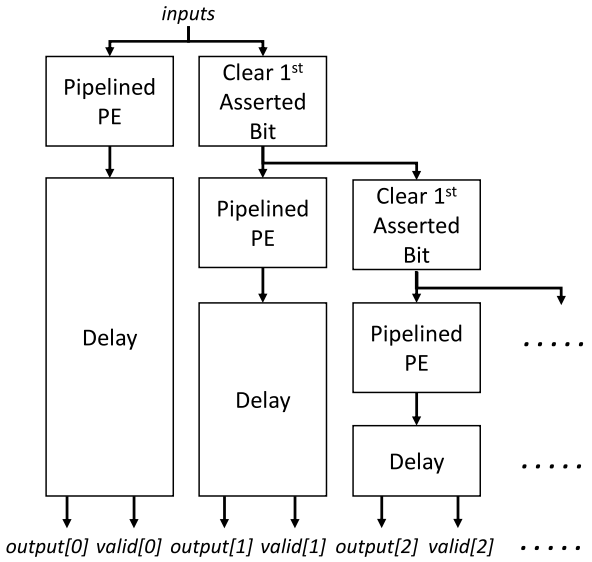


Figure 4: The pipelined multi-output priority encoder, which chains single-output pipelined priority encoders together while clearing the topmost asserted bit at each step.

bits are set equal to the lower encoder. The valid bit is asserted if either of the sub-encoder valid bits are asserted.

As illustrated in Figure 3(b) the architecture can apply this recursive rule an arbitrary number of times to create a pipelined priority encoder of any depth, where the logic delay per level remains constant for any number of inputs.

Given the pipelined, single-output priority encoder, we can now create the multi-output priority encoder as shown in Figure 4. The basic strategy is to allocate a separate single-output priority encoder for each output. However, for this strategy to work, the architecture must clear the topmost asserted bit for each additional output. To accomplish this goal, we first reverse the inputs and then leverage a bit manipulation technique that clears the lowest asserted bit. Specifically, we use the following equation:

$$input_{i+1} = input_i \text{ and } (input_i - 1) \quad (1)$$

Although not shown in the figure, this equation is optionally pipelined, which is necessary for maximizing clock frequencies. For the unpipelined results, the multi-output priority encoder's chain of subtractions between bits is the critical path.

Because each single-output priority encoder starts after the subsequent encoder, and may take multiple cycles, each encoder is followed by a delay of decreasing latency to ensure all the outputs are aligned.

4.3 Field Buffer

The architecture of the field buffer is shown in Figure 5. The field buffer has several main responsibilities: 1) store partial field data from a given input, 2) align the existing partial data with data provided by a new input, 3) when the full field has arrived, output the field along with its length. Conceptually, the field buffer acts analogously to a FIFO that can write a variable number of elements every cycle.

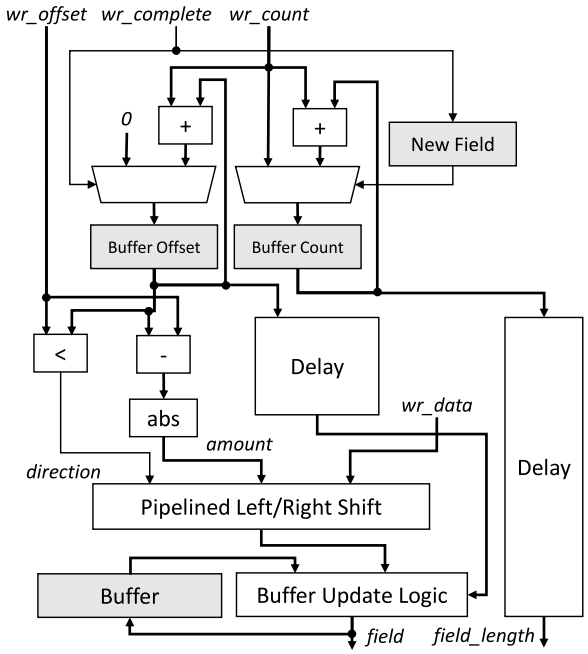


Figure 5: Overview of the field buffer, which buffers and assembles partial data into complete fields. The grey boxes are registers.

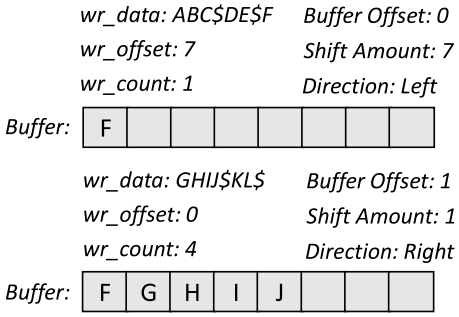


Figure 6: Examples of the internal state of the field buffer using the example from Figure 1.

The buffer's interface consists of write data (provided by delayed bus data), a write offset specifying the starting index of the field within the write data, a write count that specifies the amount of data being written, and a write complete that specifies the write contains the end of the field. There is also an implicit write enable connected to each register.

All of the alignment in the buffer is done by the *pipelined left/right shifter* (Section 4.5). The primary input to this shifter is the write data being written into the buffer. In most cases, the data for the field being written might not start at byte index 0 of the bus, so the shifter uses the write offset to account for the first index. However, the actual shift amount is not equal to this offset unless the buffer is empty. If the buffer contains partial field data from a previous input, the shifter is responsible for accounting for both the offset

within the bus, in addition to the amount of data already in the buffer (i.e., the *buffer offset*).

Figure 6 illustrates an example of this functionality for the first two inputs in Figure 1. Since the field buffer always stores partial data, the main architecture uses the field buffer to write the final F character from the first input. All the other fields are complete and do not need buffering. In this case, the write offset is 7 (the index of the character in the input) and the write count is 1. The buffer is currently empty, so the shifter would shift the input left by 7 elements, and then store the F character.

When the buffer contains data internally, the shift must take that data into consideration. For the second bus input, the main architecture writes GHIJ to complete the field. The write offset is 0 because the partial data starts at address 0 of the input. However, the buffer offset is 1 because the buffer already contains 1 element. In this case, the buffer first shifts the input data right by 1 element to align the G with the first empty buffer position. The buffer then stores the data at elements 1-4.

The *buffer update logic* is responsible for enabling the registers that need to be updated for each write. Each register in the buffer is assigned an index. The buffer update logic simply enables all registers with an index greater than or equal to the buffer offset. In addition, the actual field output does not come directly from the buffer registers, but instead from the update logic. We made this optimization to ensure that the entire parsing architecture could have a latency of 0 cycles after the final data for a field arrives on the bus. We originally had the field output come from the buffer registers, but that approach always required at least one cycle of latency, and we found the effects on clock frequency to be negligible.

Like all other components, the shifter has configurable pipelining options that can make it take any number of cycles. This pipelining creates several challenges. The field buffer needs to know how much data is stored in the buffer (i.e., the buffer offset) before doing the shift, but the data isn't actually stored into the buffer until after the shift. To handle this discrepancy, the buffer offset is updated immediately on a new write, and uses accumulation with a feedback of one cycle so that every new write knows the amount of data in the buffer by the time that data is actually written.

The buffer count is similar to the buffer offset, with one key difference. The buffer offset is reset immediately on the completion of a field, whereas the write count is reset one cycle later. This delay is needed to ensure that the field length is reported correctly.

4.4 Field Aligner

This field aligner is conceptually similar to the field buffer, with several simplifications. The biggest difference is that the field buffer needs to buffer data across inputs, whereas the field aligner is solely used for complete fields that are provided within a single input.

As a result, the field aligner contains no buffering resources, and instead only contains shifting logic. Unlike the field buffer that has to shift left or right depending on the contents of the buffer, the field aligner only ever has to shift left because there is no buffer.

Ultimately, the field aligner is a pipelined left shifter that takes a write offset based on the starting byte index of the field. The aligner also takes a write count, which it simply delays to align the field length output with the field data leaving the shifter.

4.5 Pipelined Shifter

One key component of both the field buffer and field aligner is the pipelined shifter. Although we could potentially use a traditional barrel shifter, such a shifter has massive overhead. For a shifter with n inputs, a traditional barrel shifter has a mux for each output that also has n inputs. For large values of n, the mux delay overhead and routing complexity become prohibitive. One alternative is to pipeline the muxes across multiple cycles, but that does not reduce the prohibitive routing complexity.

Existing strategies pipeline a large barrel shifter into multiple stages where each stage uses muxes with only 2 inputs along with an optional register, which we refer to as the *binary-decision shifter*. For example, for a 16-input barrel shifter, the first stage shifts by 8 or 0, the second by 4 or 0, the third by 2 or 0, and the fourth by 1 or 0.

One problem with the binary-decision shifter is that it greatly increases the latency of the shift, due to passing through $\lceil \log_2(n) \rceil$ stages, which can be prohibitive for latency-sensitive designs. In addition, 2-input muxes do not fully utilize LUTs on some FPGAs, where larger numbers of inputs can map to the same number of resources.

We solve this problem by expanding the binary-decision shifter to support decisions of any power-of-two base, which translates to power-of-two inputs to each multiplexor. This optimization was motivated by the observation that some newer FPGAs are optimized to handle wide muxes. For example, the UltraScale+ can implement a 32-input mux in a single CLB [21].

To explain our optimized shifter, we use b to represent the base number of mux inputs, where the select line has $x = \lceil \log_2(b) \rceil$ bits. The shifter architecture performs the shift across $\lceil n/x \rceil$ stages, where n is the number of inputs to the shifter. In each stage i of the shifter, each mux j selects from b muxes from the preceding stage, using the following equation:

$$f(s) = s \cdot 2^{b-i \cdot x} + j, \quad \text{for } s \in \{0, 1, 2, \ldots, b-1\} \tag{2}$$

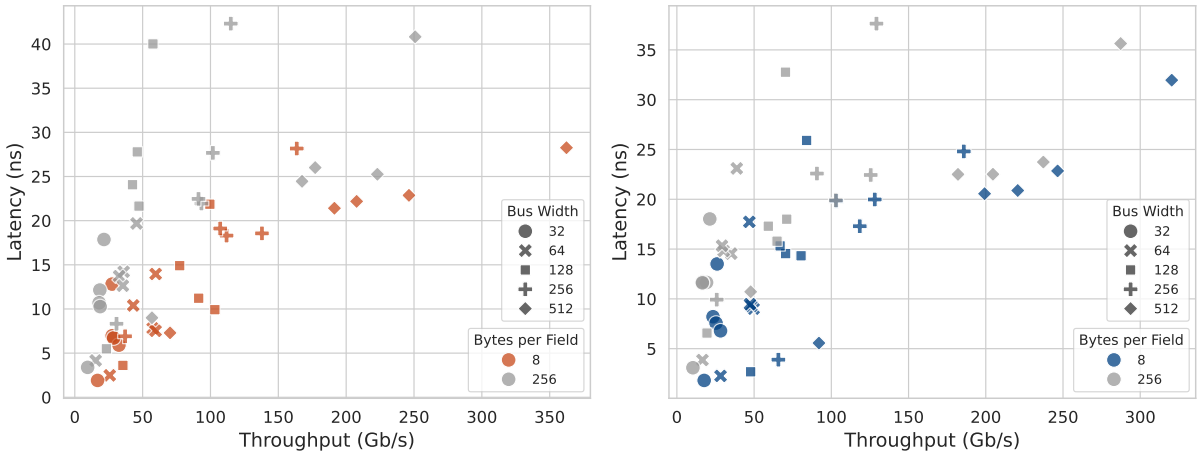
For example, for a 32-input shifter ($n = 32$), to optimize for muxes with 4 inputs ($b = 4$), the first stage of the shifter ($i = 1$) would have muxes that select values from stage 0 with an offset of 0, 8, 16, and 24 for the mux position. Stage 2 muxes would select between 0, 2, 4, and 6. For $b = 8$, stage 1 muxes would select from 0,4,8,12,16..28.

5 EXPERIMENTAL RESULTS

This section is organized as follows. We first describe the experimental setup in Section 5.1. We then evaluate the trade-offs between latency and throughput across a wide range of configuration options in Section 5.2. Finally, Section 5.3 evaluates the scalability of the design for different maximum numbers of fields per bus input.

5.1 Experimental Setup

We designed the parsing architecture using 3,872 lines of SystemVerilog, including testbenches. To demonstrate effectiveness on multiple FPGAs, we evaluate both AMD and Intel FPGAs. For the AMD experiments, we synthesized the designs using Vivado 2023.1 targeting a Virtex Ultrascale+ xcvu9p-flga2104-2L-e. For the Intel experiments, we synthesized using Quartus Prime Pro 23.1.0.115 targeting an Agilex AGIC040R39A2I3V. We do not have physical access to boards with either of these FPGAs, but we have verified correct



(a) AMD Virtex Ultrascale+ **(b) Intel Agilex**

Figure 7: A comparison of trade-offs between latency and throughput for different bus widths, maximum bytes per field, and pipelining options for (a) AMD Virtex Ultrascale+ and (b) Intel Agilex. These results assume a maximum number of fields per input of 8.

functionality on an Stratix 10 provided by an Intel Platform Acceleration Card (PAC). We do not report the Stratix 10 results due to it being an older technology. Our intent is not to compare devices capabilities, but to demonstrate efficient operation on different devices.

We created the RTL code with parameters for the bus width, maximum number of bytes per field, maximum number of fields per bus input, and pipelining configuration options that include customizing priority encoders and muxes to the specific LUT architecture of each FPGA. These pipelining options also offer the flexibility to optionally insert registers between each level of the synthesized LUT hierarchy.

To conduct design-space exploration, we developed a set of Python scripts that take a YAML file as input to define the parameter combinations to explore. Initially, we used these scripts to validate functionality via simulation for thousands of parameter combinations. Subsequently, we employed similar scripts to synthesize, place, and route over 100 combinations across both FPGAs. The synthesis exploration was limited by placement and routing times, which took approximately one week cumulatively across all experiments.

For placement and routing, we applied a uniform clock frequency constraint of 1 GHz. Ideally, we would have explored different constraints for each parameter combination, but such exploration would have required a prohibitive amount of time. We estimate that the designs with low clock frequencies are pessimistic because, in our experience, using a constraint that significantly exceeds an attainable frequency results in lower-quality placement and routing.

To determine the maximum clock frequency of each configuration, we leveraged Vivado's out-of-context (OOC) flow, where we constrained the placement of the clock buffer for realistic clock skew. For Intel results, we used virtual pins for all I/O except for the clock and reset. For both vendors, we registered all I/O except for the clock and reset.

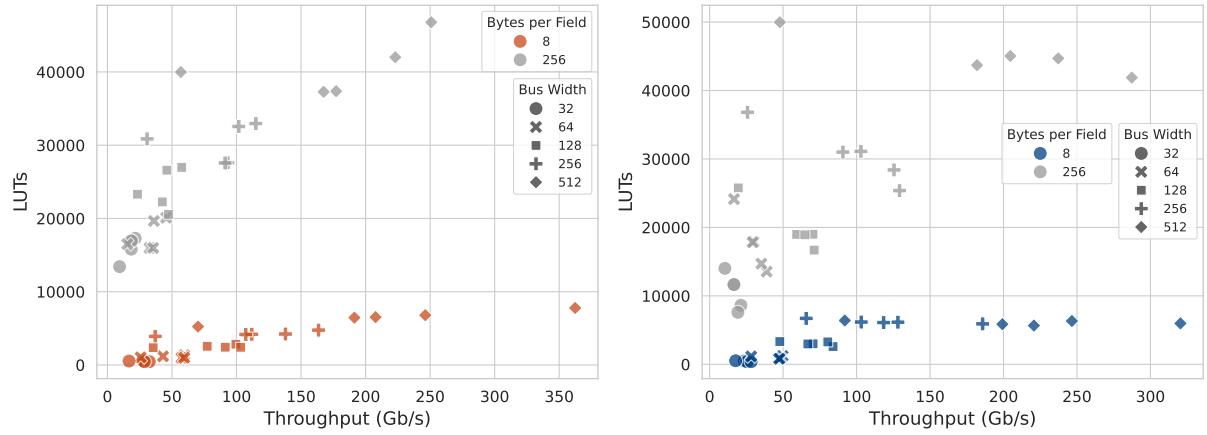
5.2 Throughput vs. Latency

In this section, we compare trade-offs between latency and throughput for different bus widths, maximum bytes per field, and different pipelining options. Figure 7 compares the trade-offs on both the AMD Virtex Ultrascale+ and Intel Agilex. The different shapes for each point correspond to different bus widths, and the different colors correspond to different maximum bytes per field. We consider two different maximum bytes per field: 8 (for an HFT use case) and 256 (for general strings). All experiments used a maximum number of fields per input of eight. The pipelining options are not explicitly labeled due to requiring too many dimensions to visualize, but each separate point within a given series is a different pipelining option.

For the AMD Virtex Ultrascale+, Fig. 7 shows a wide range of trade-offs. As expected, increasing throughput (usually with more registers) increased latency, and reducing latency decreased throughput. However, when just considering the Pareto-optimal solutions (i.e., the Pareto Frontier) across all configurations, the Pareto-optimal points yielded minor sacrifices to latency for significant improvements in throughput. The experiments with 256 maximum bytes per field generally showed a wider range of trade-offs, but interestingly, the Pareto Frontier was reasonably similar to the experiments using 8 maximum bytes per field. This frontier similarity suggests that the architecture scales well to larger maximum numbers of bytes per field.

With the exception of one configuration, all designs were able to achieve a throughput of 10 Gb/s. When using a 32-bit bus with no pipelining and 8 maximum bytes per field, the latency was only 1.9 ns. To our knowledge, this is the lowest latency for a variable-length field parser. Even compared to fixed-length parsing studies [3], the latency of our architecture was over 10x smaller.

At a throughput of 40 Gb/s, the minimum latency was 7.3 ns, which was achieved by a 512-bit bus with 8 maximum bytes per field and no pipelining. At a throughput of 100 Gb/s, the minimum



(a) AMD Virtex Ultrascale+ **(b) Intel Agilex**

Figure 8: A comparison of throughput and LUT usage for different bus widths, maximum bytes per field, and pipelining options for (a) AMD Virtex Ultrascale+ and (b) Intel Agilex. These results assume a maximum number of fields per input of 8.

latency was 9.9 ns, which used a 128-bit bus, 8 maximum bytes per field, and an amount of pipelining that generally skipped two levels of LUTs. The highest throughput design achieved 362 Gb/s at a latency of 28.3 ns, using a 512-bit bus with all pipelining options enabled.

The Intel Agilex experiments showed the same trends as the AMD Virtex Ultrascale+, with similar trade-offs across most experiments. For 10 Gb/s, the minimum latency was 1.8 ns for a 32-bit bus, 8 maximum bytes per field, and no pipelining. For 40 Gb/s, the minimum latency was 2.7 ns for a 128-bit bus with 8 maximum bytes per field, and no pipelining. For 100 Gb/s, the minimum latency was 19.8 ns for a bus width of 256, 8 maximum bytes per field, and pipelining that skipped two levels of LUTs. Maximum throughput was 320 Gb/s for a 512-bit bus, 8 maximum bytes per field, and full pipelining.

For the AMD results, the average clock frequencies for 32-bit, 64-bit, 128-bit, 256-bit, and 512-bit buses were 684 MHz, 641 MHz, 487 MHz, 387 MHz, and 381 MHz, respectively. The maximum frequencies were 1016 MHz, 931 MHz, 806 MHz, 639 MHz, and 708 MHz, respectively, which occurred for 8 bytes per field. Note that some of these high frequencies result from "out-of-context" analysis and would be restricted by device limitations.

For the Intel results, the average clock frequencies for 32-bit, 64-bit, 128-bit, 256-bit, and 512-bit buses were 640 MHz, 579 MHz, 496 MHz, 420 MHz, and 398 MHz, respectively. The maximum frequencies were 883 MHz, 776 MHz, 656 MHz, 726 MHz, and 626 MHz, respectively, which again occurred for 8 bytes per field.

Figure 8 illustrates how the LUT usage varies with throughput for different bus widths, maximum bytes per field, and pipelining options. All experiments again assume a maximum number of fields per bus input of eight.

For a maximum bytes per field of 8, both the AMD and Intel experiments demonstrate a clear linear relationship between throughput and LUTs. For all bus widths, the LUT usage is small for any modern FPGA, averaging just 6,560 and 5,507 LUTs for the 512-bit bus on AMD and Intel, respectively. Both of these use less than 1% of available FPGA resources.

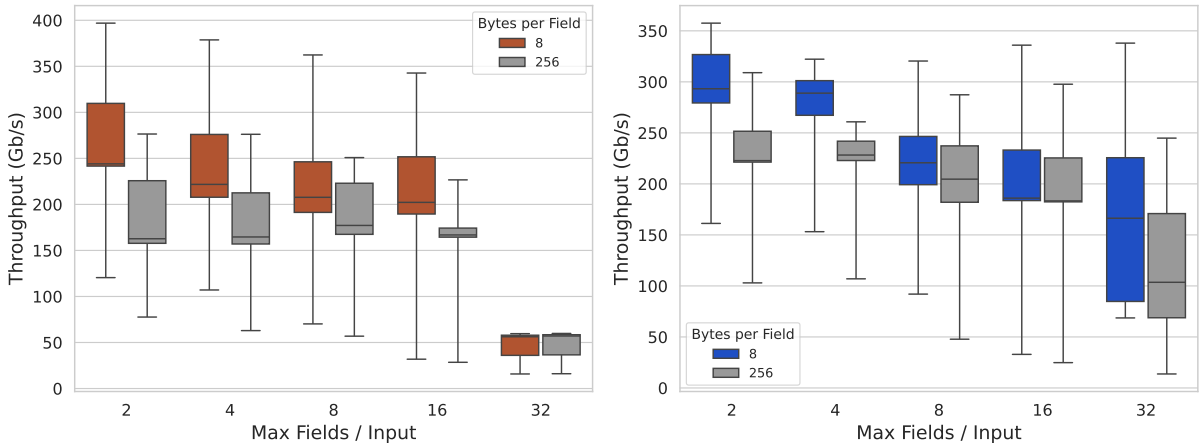
For a maximum bytes per field of 256, there is still a linear relationship between throughput and LUTs, but with a higher slope. For the AMD results, the Pareto Frontier demonstrates 100 Gb/s can be achieved with 31,305 LUTs. The Intel results achieve 100 Gb/s at a slightly lower utilization of 25,383 LUTs. For a 1 million LUT FPGA, these LUTs represent only 3.1% and 2.5% of avaliable LUTs. 40 Gb/s was achieved with 20,102 and 18,999 LUTs, respectively. 10 Gb/s was achieved with 15,803 and 7,594 LUTs, respectively.

5.3 Scalability

Figure 9 evaluates the scalability of the architecture for a 512-bit bus by evaluating how throughput changes for increasing amounts of maximum fields per bus input, up to the maximum possible value of 32 for a 512-bit bus. We use a box plot to show the distribution of all configuration options that we explored. However, the most important results for scalability are the maximum and median values.

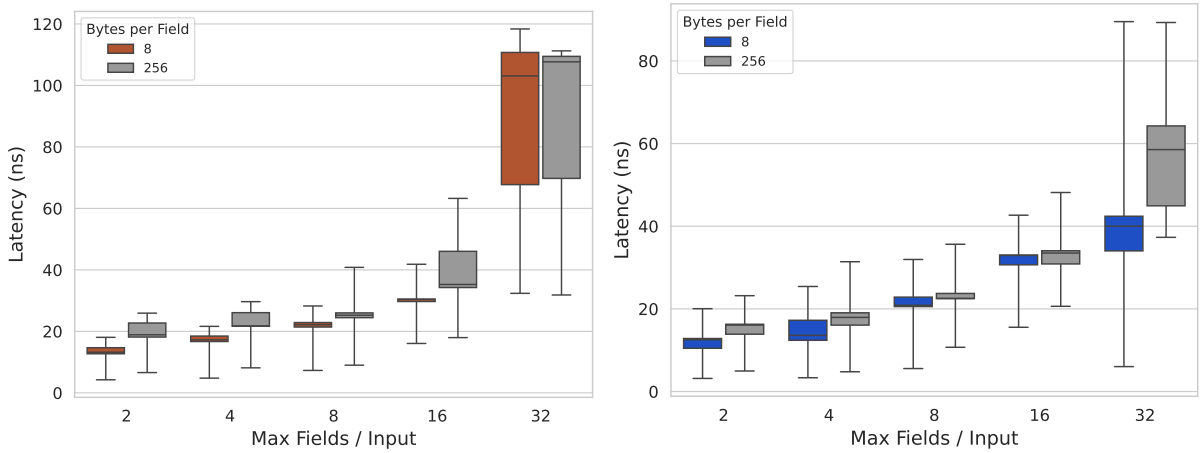
It is important to note that 32 and 16 fields per input are stress tests that would never occur in a realistic application. 32 and 16 fields per input on a 512-bit bus would consist entirely of 1-byte and 2-byte fields, respectively. 8 fields per input could be representative of an application with many small strings, such as ticker symbols in high-frequency. Most actual protocols are a mixture of variable and fixed-length fields, in which case 2-4 fields per input would be realistic.

For both AMD and Intel, the results showed good throughput scalability up to 16 fields per input. For the AMD results, the maximum throughput decreased linearly with a low slope. For the Intel results, maximum throughput stayed relatively constant. For 32 fields per input, the AMD results demonstrated a significant decrease in throughput. By contrast, the Intel results showed continued scalability. We suspect this improved scalability was due to



(a) AMD Virtex Ultrascale+ (b) Intel Agilex

Figure 9: Throughput of all configuration options for different maximum bytes per field and maximum fields per bus input. All results use a 512-bit bus.



(a) AMD Virtex Ultrascale+ (b) Intel Agilex

Figure 10: Latency of all configuration options for different maximum bytes per field and maximum fields per bus input. All results use a 512-bit bus.

the HyperFlex registers. However, as stated earlier, 32 fields per input is a stress test that would likely not occur in real applications.

Figure 10 evaluates how latency changes for increasing amounts of maximum fields per input. For both AMD and Intel experiments, the minimum and median latencies increased linearly, again up to 16 fields per input. At 32 fields per input, the minimum latency increased slightly, but the median latency of all evaluated points increased significantly, with the exception of the Intel results for 8 bytes per field.

Figure 11 demonstrates the average clock frequency across all pipelining configurations for the experiments in the previous figure. In most cases, clock frequency had approximately a linear decrease with increased maximum fields per input, which suggests good scalability. Average clock frequency fell below 100 MHz for 32 fields per input and 256 maximum bytes per field, but as mentioned earlier, 32 fields per input on a 512-bit bus is an extreme stress test. Even at 16 fields per input, which is still highly pessimistic, average clock frequencies were always above 300 MHz.

6 CONCLUSIONS

In this paper, we introduced an FPGA accelerator for achieving line-rate processing of variable-length fields. The accelerator is parameterized with configuration options to enable different trade-offs between latency and throughput, in addition to logic optimizations for each targeted FPGA. We demonstrated attractive trade-offs for common SmartNIC use cases, achieving latencies of 1.8 ns at 10

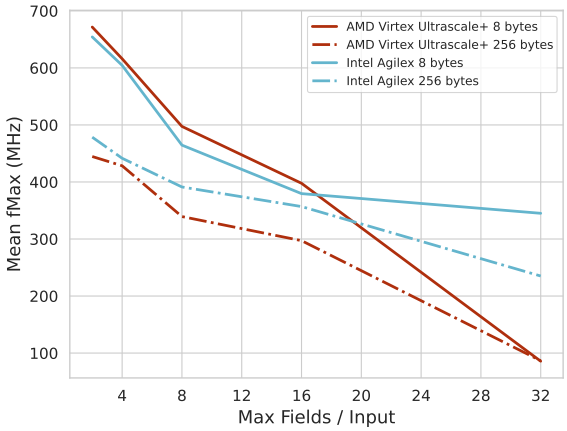


Figure 11: Average clock frequencies for a 512-bit bus, for different maximum fields per bus input and maximum bytes per field, on both AMD Virtex Ultrascale+ and Intel Agilex.

Gb/s, 2.7 ns at 40 Gb/s, and 9.9 ns at 100 Gb/s. To our knowledge, these latencies are the fastest of any known study.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1718033 and CCF-1909244.

REFERENCES

[1] 2022. Financial Information eXchange (FIX) Protocol. Online. https://www.fixtrading.org/online-specification/

[2] Srinivas Aluru and Nagakishore Jammula. 2014. A Review of Hardware Acceleration for Computational Genomics. *IEEE Design Test* 31, 1 (2014), 19–30. https://doi.org/10.1109/MDAT.2013.2293757

[3] Marc Battyani. 2021. A sub 25 nanoseconds Open Source NASDAQ ITCH FPGA Parser. https://github.com/mbattyani/sub-25-ns-nasdaq-itch-fpga-parser#a-sub-25-nanoseconds-open-source-nasdaq-itch-fpga-parser. Accessed: October 11, 2023.

[4] Andrew Bitar, Mohamed S. Abdelfattah, and Vaughn Betz. 2015. Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA. In *2015 International Conference on Field Programmable Technology (FPT)*. 24–31. https://doi.org/10.1109/FPT.2015.7393125

[5] Gordon Brebner and Weirong Jiang. 2014. High-Speed Packet Processing using Reconfigurable Computing. *IEEE Micro* 34, 1 (2014), 8–18. https://doi.org/10.1109/MM.2014.19

[6] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2022. HXDP: Efficient Software Packet Processing on FPGA NICs. *Commun. ACM* 65, 8 (jul 2022), 92–100. https://doi.org/10.1145/3543668

[7] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[8] Danilo Cerović, Valentin Del Piccolo, Ahmed Amamou, Kamel Haddadou, and Guy Pujolle. 2018. Fast Packet Processing: A Survey. *IEEE Communications Surveys Tutorials* 20, 4 (2018), 3645–3676. https://doi.org/10.1109/COMST.2018.2851072

[9] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 51–64.

[11] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '12)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/2145694.2145704

[12] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[13] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/3020078.3021745

[14] Ziyi Lv and Jing Zhang. 2022. A Survey of FPGA-Based Deep Learning Acceleration Research. In *The International Conference on Image, Vision and Intelligent Systems (ICIVIS 2021)*, Jian Yao, Yang Xiao, Peng You, and Guang Sun (Eds.). Springer Nature Singapore, Singapore, 59–65.

[15] NVIDIA. 2023. *NVIDIA DOCA GPU Packet Processing Application Guide*. https://docs.nvidia.com/doca/sdk/gpu-packet-processing/index.html [Accessed: 10/13/2023].

[16] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 531–548. https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[17] Roberto Sierra, Filippo Mangani, Carlos Carreras, and Gabriel Caffarena. 2019. High-Performance Decoding of Variable-Length Memory Data Packets for FPGA Stream Processing. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 307–313. https://doi.org/10.1109/FPL.2019.00056

[18] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 387–398. https://doi.org/10.1109/HPCA.2019.00052

[19] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. 1078–1086. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199

[20] David Wills. 2023. *Fast Track Data Center Workloads and AI Applications with NVIDIA DOCA 2.2*. https://developer.nvidia.com/blog/fast-track-data-center-workloads-and-ai-applications-with-nvidia-doca-2-2/ [Accessed: 10/13/2023].

[21] Xilinx, Inc. 2017. *Xilinx UltraScale Architecture Configurable Logic Block*. https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb Accessed: October 10, 2023.