Fleet: A Framework for Massively Parallel Streaming on FPGAs

James Thomas
Stanford University
jjthomas@stanford.edu

Pat Hanrahan
Stanford University
hanrahan@cs.stanford.edu

Matei Zaharia
Stanford University
matei@cs.stanford.edu

Abstract

We present Fleet, a framework that offers a massively parallel streaming model for FPGAs and is effective in a number of domains well-suited for FPGA acceleration, including parsing, compression, and machine learning. Fleet requires the user to specify RTL for a processing unit that serially processes every input token in a stream, a far simpler task than writing a parallel processing unit. It then takes the user's processing unit and generates a hardware design with many copies of the unit as well as memory controllers to feed the units with separate streams and drain their outputs. Fleet includes a Chisel-based processing unit language. The language maintains Chisel's low-level performance control while adding a few productivity features, including automatic handling of ready-valid signaling and a native and automatically pipelined BRAM type. We evaluate Fleet on six different applications, including JSON parsing and integer compression, fitting hundreds of Fleet processing units on the Amazon F1 FPGA and outperforming CPU implementations by over 400× and GPU implementations by over 9× in performance per watt while requiring a similar number of lines of code.

CCS Concepts. • **Hardware → Reconfigurable logic applications**; **Hardware description languages and compilation**; **Application specific processors**.

Keywords. FPGAs, HDLs, streaming, identical processors

ACM Reference Format:
James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3373376.3378495

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378495

1 Introduction

As FPGAs have become available in datacenters [9] and public clouds [1, 2], the question of how to easily program and use them has become increasingly important. Over a decade ago, the same question arose for distributed networks of multicore CPUs, and in the intervening years a number of distributed computing frameworks have become popular, including Spark [21] and MapReduce [11]. The key insight of these systems is to have the user only write serial code to process a single stream of inputs, with the system automatically running copies of this code on many streams across many cores in a cluster. This approach spares the user the challenge of writing multicore or distributed code. Unfortunately, such a computation model has not yet been developed for FPGAs. Even high-level frameworks like C-based HLS [10] and others that expose a functional programming model [14, 16] expect users to parallelize the processing of a single stream.

There is a need for an FPGA programming framework where users write a serial processing unit to process a single stream of data, and the framework then replicates the unit many times and generates memory controllers to feed each unit with its own stream of data and store its output. Writing serial hardware should be easier for users than writing parallel hardware, which often requires algorithmic changes and complex logic to distribute inputs to different computational units. We believe that such a framework, designed for what we call *multi-stream parallelism* or *massively parallel streaming*, can be applied to a wide variety of big data problems, including parsing, compression, string search, and machine learning.

This paper presents Fleet, a framework designed for multi-stream applications on FPGAs. Fleet requires users to provide RTL with a ready-valid interface that serially processes a single stream of tokens. This RTL can be generated by higher-level tools, and Fleet provides a Scala-embedded language that is an extension of Chisel [8] with a few productivity-ehancing features for writing these processing units. In particular, the Fleet language automatically handles ready-valid signaling and provides an automatically pipelined BRAM type. These features eliminate rote tasks for the developer without compromising the low-level performance control offered by Chisel. Once a processing unit is written, the Fleet framework provides high developer productivity by automatically generating as many copies of the unit as the user

RIGHTSLINK

wants and a soft memory controller to feed and drain the processing units. This productivity does not come at the cost of performance: multi-stream applications written in Fleet for a modern FPGA outperform CPU and GPU implementations in performance per watt and have better memory performance and processing unit throughput than HLS implementations.

The Fleet language requires that there are no dependent BRAM reads and each BRAM is read at most once and written at most once in each processing step, or virtual cycle. Since accesses to BRAMs are restricted in this way, the Fleet compiler can always generate a two-stage pipeline for virtual cycles, with one stage for BRAM reads and one stage for writes. In contrast, the performance of code generated by C-based HLS systems is far less predictable, as the compiler is required to perform fickle analyses to determine whether different BRAM accesses in a program conflict and whether extra pipeline stages must be inserted.

The Fleet compiler generates many copies of the specified processing unit and a high-performance memory controller that would be challenging for users to write manually. Our round-robin input and output controllers take advantage of the predictable, sequential memory access patterns of the individual processing units. Our input controller requests the next block of data for each processing unit well before it is required to avoid memory latency-induced delays, and feeds input blocks in parallel to multiple processing units to overcome the limited input read throughput of individual units. Our output controller behaves in a symmetric manner.

We conduct an extensive evaluation of Fleet using six applications, including a JSON parser, a compressor for integers, a Bloom filter constructor, and a regex matcher, all written in the Fleet language. To demonstrate that FPGAs can be an excellent platform for multi-stream applications with the right programming framework, we compare our Fleet applications running on the Amazon F1 platform to CPU and GPU implementations. We are able to fit hundreds of stream processing units on the F1 FPGA and show over 400× and 9× improvements in performance per watt compared to the CPU and GPU, respectively. The improvements in performance over CPU and GPU are due primarily to fusion of multiple CPU/GPU instructions into one cycle and the high cost of control flow divergence across streams on the CPU/GPU. Further, the total number of lines of Fleet code required to achieve this performance for each application is comparable to that for the CPU/GPU implementations. We show that our input memory controller can read data from the four DDR3 memory channels on the Amazon F1 and feed it to the processing units at a rate of 27.24 GB/s, which is 91% of the peak memory throughput at 125 MHz. Finally, we perform an evaluation of a commercial HLS system, showing that it generates a memory controller with 10× lower performance than ours, drastically limiting its maximum throughput, and that its OpenCL language abstraction often results in long and inefficient processing unit pipelines.

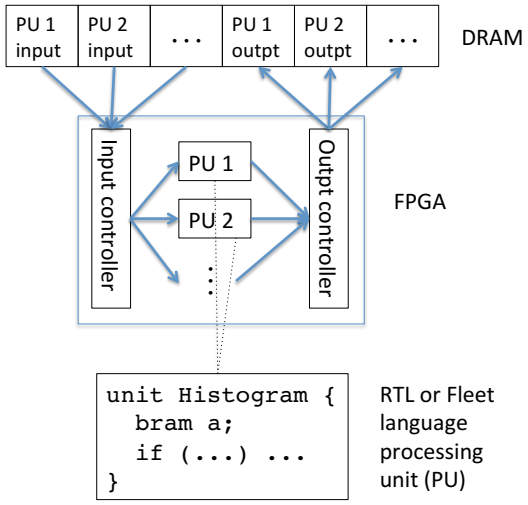

Figure 1. Fleet framework overview

In summary, our contributions are the following:

1. The design of a framework for multi-stream applications that allows developers to process tens of gigabytes of data per second on modern FPGAs
2. A Chisel-based language for stream processing units that provides a convenient token-based processing abstraction, automatic handling of ready-valid signaling, and an automatically pipelined BRAM type (Sections 3 & 4),
3. A high-performance memory controller to drive the processing units (Section 5)
4. An evaluation on several important applications on the Amazon F1 platform, demonstrating significant performance-per-watt improvements over CPU and GPU implementations at little cost to developer productivity, and better memory system and processing unit performance than a commercial HLS system (Section 7)

2 System and API Overview

Fleet provides a language for users to specify the processing of a single stream of data. Programs in the language describe the "virtual cycles," consisting of state updates and output emissions, to be executed for every input token. High performance is achieved by synthesizing many copies of this processing unit on the target FPGA, along with a memory controller that supplies input tokens and drains output tokens from the units. Each processing unit operates on a separate stream. Figure 1 shows an overview of the Fleet framework.

To use a generated Fleet design that has been loaded onto an FPGA, the user must fill a contiguous buffer with the streams for each processing unit in software. The Fleet software runtime transfers the buffer to FPGA DRAM. It then

RIGHTSLINK

Chisel-like registers, binary/unary operators, and conditional blocks	
Native BRAM type with restricted read/write per virtual cycle	
`input` keyword to access current input token	
`emit` operator to produce output token	
`while` loop to take multiple virtual cycles for current input token	

Figure 2. Fleet language features

kicks off the processing units, with each processing unit directed to process a different stream. Each processing unit writes its output to its own region of a contiguous buffer. Once all of the processing units have finished processing their input streams, the output buffer is transferred back to host DRAM.

To use Fleet, users must have a way to split up a large input into many smaller streams that can be processed in parallel. This is a reasonable expectation: Spark and MapReduce have the same requirement on their inputs. There are often fast routines to split a single large input. For example, in files containing JSON records, individual records are often separated by newlines [18], so files can be split easily by a fast, vectorized newline finder on the CPU. In some cases, such as in string search applications, a single input file can be split at arbitrary points, with a small amount of extra processing performed on the CPU to find matches at boundaries. The input streams to the different processing units should be roughly similar in size to ensure high throughput, as the system cannot load balance across processing units since each unit may have state specific to its own stream.

3 The Fleet Language

Fleet allows users to specify a single serial stream processing unit in any standard RTL language (Verilog, VHDL, etc.) accepted by the target FPGA's synthesis tools, as long it has the ready-valid IO interface described at the start of Section 4. However, we have observed that manually managing ready-valid signaling (discussed more in Section 4) and pipelining accesses to BRAMs is quite tedious and error-prone, so we provide a processing unit language that automatically handles these rote tasks. The language takes care not to sacrifice the low-level performance control offered by RTL. In particular, the Fleet language does not offer more powerful HLS-style optimizations like automatic logic pipelining and resource multiplexing, which may not behave exactly as the user wants.

The Fleet language is a Scala-embedded DSL for specifying the behavior of a single serial stream processing unit. It is an extension of Chisel [8] for token-based streaming, with the user's Chisel executed repeatedly for each input token. Our knowledge that the logic is repeatedly executed allows us to introduce a native BRAM type and automatically pipeline reads and writes to it, as long as the logic does not

```
1  unit BlockFrequencies(inputTokenSize=8,
       outputTokenSize=8) {
2    itemCounter := reg(bits=7, init=0)
3    frequencies := bram(elements=256,
         bitsPerElmt=8)
4    frequenciesIdx := reg(bits=9, init=0) // 9
         bits to store largest value 256
5
6    if (itemCounter == 100) { // emit
         frequencies
7      while (frequenciesIdx < 256) {
8        emit(frequencies[frequenciesIdx])
9        frequencies[frequenciesIdx] = 0
10       frequenciesIdx += 1
11     }
12     frequenciesIdx = 0
13   }
14   // process current input token
15   frequencies[input] += 1
16   itemCounter = itemCounter == 100 ? 1 :
         itemCounter + 1
17 }
```

Figure 3. Frequency-counting Fleet processing unit

perform more BRAM accesses per cycle than supported by the underlying technology.

The state elements in the Fleet language are registers, vector registers with random access support, and BRAMs, all with user-specified bitwidths. A wire type is available to hold temporary values. The statements in the language include assignments to state elements and `emit`s of output tokens, and statements can be contained in `if`, `else if`, and `else` blocks. As in Chisel, all statements are evaluated concurrently. Basic integer arithmetic, Boolean, and bitwise operators are provided, and an expression called `input` provides access to the current input token. The bitwidths of input and output tokens are fixed and must be defined at compile time. After all input tokens have been processed, the user's logic is run with a dummy input token to perform any necessary cleanup, with the `stream_finished` identifier set to true. There is no way to access DRAM outside of the incoming and outgoing streams of tokens; this simplifies our memory controller design without hurting the expressibility of our applications.

A very simple Fleet language processing unit is as follows:

```
unit Identity(inputTokenSize=8, outputTokenSize=8)
    {
  if (!stream_finished) emit(input)
}
```

The processing unit declaration specifies that the input bitstream should be broken up into 8-bit tokens and that the logic should be fired once for every 8-bit token. It also specifies that any output tokens produced by the logic will be 8

RIGHTSLINK

bits in size. The processing unit logic simply emits the input stream back to the output unmodified.

Figure 2 summarizes the features of the Fleet language. Figure 3 shows an example processing unit for computing and emitting a 256-element histogram (or frequency count) for each block of 100 8-bit input tokens. We refer to it in this section and the next. The example assumes that the `frequencies` BRAM starts zero-initialized, which is the case on most FPGAs. Due to the `stream_finished` execution of the logic after the stream is complete, the histogram for the final input block is correctly produced.

To preserve the low-level performance control of Chisel, a Fleet user is guaranteed that each execution of their token processing logic, including BRAM operations, will occur in a single cycle. However, real FPGA BRAMs support limited reads and writes per cycle and have one cycle of latency to retrieve or store data after a read or write address is supplied [3, 5, 6]. Since BRAMs cannot actually service reads and writes in a single cycle, we call each execution of the user's token processing logic a *virtual cycle*. In order to ensure that we can pipeline BRAM accesses and maintain our guarantee of a throughput of one virtual cycle per real cycle, we restrict the use of BRAMs in certain ways.

First, dependent BRAM reads are not allowed in a virtual cycle, as these would take more than one real cycle to resolve and cannot always be pipelined. Example expressions where the output of BRAM `a` depends on the output of BRAM `b` include `a[b[0]]` and `if (b[0]) x = a[0] else x = a[1]`, both of which require two real cycles to produce the output of `a`.

Furthermore, Fleet programs can only read a BRAM at one address and write it at one address in each virtual cycle, which matches the capabilities of the underyling technology. Technology BRAMs sometimes support performing two reads or two writes in a single cycle; our current restrictions on Fleet BRAM accesses prevent us from using this feature, which was not important for our applications' performance.

The final restriction in the language is on `emit` operations. Only one token can be emitted per virtual cycle. If this were not the case, the system would have no way to order the multiple emitted tokens in the output stream, since Fleet programs have concurrent semantics.

Adherence to the language restrictions can be checked by our software simulator, which runs a Fleet program on an input stream and detects dependent BRAM reads, multiple BRAM reads or writes in a single virtual cycle, and multiple `emits` in a single virtual cycle. Although we have not implemented one, a static analyzer could also guarantee that certain well-structured programs do not violate the restrictions, or we could insert logic to perform runtime checks.

Due to the restrictions on BRAM reads and writes and `emits`, it may not always be possible to perform all necessary processing for an input token in a single virtual cycle. As a result, the Fleet language features a `while` loop construct. A `while` loop has a condition, and virtual cycles executing the loop body will run until the condition is false, without advancing the input token. After the `while` loop is complete, a final virtual cycle will execute statements outside of the loop, and then the input token will be advanced. In the example in Figure 3, if the `while` loop is active (`itemCounter == 100`), 256 virtual cycles running lines 8-10 are executed to emit the 256 entries in the frequency array and clear them to zero. At the end of the loop (when `frequenciesIdx == 256`), a final virtual cycle is run that executes lines 12 and 15-16, consuming the current input token. If the `while` loop is not active, the input token is consumed in a single virtual cycle that executes lines 15-16.

If multiple `while` loops appear in an Fleet program, loop virtual cycles are executed until all `while` conditions become false. Nesting of loops is not currently supported; nested loops can be transformed into a single unnested loop with additional state machine states. Our example programs were all fairly easy to express without nested loops.

As in Chisel, it is possible to implement multi-stage token processing pipelines in the Fleet language. The user must explicitly define the register state and control logic for each pipeline stage, and implement optimizations such as result forwarding across stages. `while` loops can be used to implement bubbles where multiple cycles are required before a new input token can enter the pipeline.

As in Chisel, embedding the Fleet language in Scala improves productivity by allowing users to write Scala code that generates Fleet statements. This is particularly useful for parameterized stream processing units whose size or behavior is defined by compile-time parameters.

In summary, the Fleet language allows users to define stream processing units, providing a token-at-a-time processing abstraction and automatically pipelined BRAM access. Despite its productivity features, Fleet gives the user full control over and understanding of performance, and is general enough to express arbitrary streaming pipelines.

4 Compiling the Fleet Language

In this section we describe the compilation of a single Fleet language processing unit to RTL. The compiler spares the programmer the challenge of generating ready-valid IO signals. It also automatically pipelines BRAMs while guaranteeing high throughput. In particular, because of the restrictions the Fleet language imposes on BRAM use in a virtual cycle, the compiler can always generate a virtual cycle pipeline that runs at a throughput of one virtual cycle per real cycle in the absence of input and output stalls. In contrast, compilers for HLS C and other languages without these restrictions can generate arbitrarily slow pipelines if pragmas are not carefully used, reducing performance predictability and control for users.

RIGHTSLINK

```
input input_token[7:0], input_valid, output_ready, input_finished;
output input_ready, output_token[7:0], output_valid, output_finished;

reg[7:0] i; // stores current input token
reg v = 0; // whether the virtual cycle is currently executing
reg f = 0; // whether we have started the stream_finished virtual cycle

reg[6:0] itemCounter = 0;
bram frequencies; // 8 data bits, 8 address bits
reg[8:0] frequencies_lastAddr = -1; // extra bit for sentinel value
reg[7:0] frequencies_lastData;
reg[8:0] frequenciesIdx = 0;

wire v_done = v && (!output_valid || output_ready); // whether current virtual cycle is finishing
wire while_done = !(itemCounter == 100 && frequenciesIdx < 256);

wire frequenciesIdx_n = (itemCounter == 100 && frequenciesIdx < 256) ? frequenciesIdx + 1 : ((
    itemCounter == 100 && while_done) ? 0 : frequenciesIdx);
wire itemCounter_n = while_done ? (itemCounter == 100 ? 1 : itemCounter + 1) : itemCounter;
if (v_done) {
  frequenciesIdx <= frequenciesIdx_n;
  itemCounter <= itemCounter_n;
  if (frequencies.wr_en) {
    frequencies_lastAddr <= frequencies.wr_addr;
    frequencies_lastData <= frequencies.wr_data;
  }
}

wire frequencies_cur_rd_addr = (itemCounter == 100 && frequenciesIdx < 256) ? frequenciesIdx : i;
wire frequencies_next_rd_addr = (itemCounter_n == 100 && frequenciesIdx_n < 256) ? frequenciesIdx_n :
    input_token;
frequencies.rd_addr = v_done ? frequencies_next_rd_addr : frequencies_cur_rd_addr;
wire frequencies_rd_data = (frequencies_cur_rd_addr == frequenciesLastAddr) ? frequencies_lastData :
    frequencies.rd_data;

frequencies.wr_en = v_done && ((itemCounter == 100 && frequenciesIdx < 256) || while_done);
frequencies.wr_addr = (itemCounter == 100 && frequenciesIdx < 256) ? frequenciesIdx : i;
frequencies.wr_data = (itemCounter == 100 && frequenciesIdx < 256) ? 0 : frequencies_rd_data;

input_ready = !v || (while_done && (!output_valid || output_ready));
output_token = frequencies_rd_data;
output_valid = v && (itemCounter == 100 && frequenciesIdx < 256);
if (input_ready) {
  v <= input_valid || (!f && input_finished);
  f <= f || input_finished;
  i <= input_token;
}
output_finished = !v && f;
```

Figure 4. Generated RTL for the Fleet processing unit in Figure 3, handling BRAM pipelining and IO/`while` stalls for the user

Figure 4 shows the RTL output produced by the compiler on the histogram example in Figure 3, and we refer to it throughout this section. Each processing unit has the following IO interface to the memory controller, which is described in more detail in Section 5:

```
input input_token // multiple bits
input input_valid
output input_ready
output output_token // multiple bits
output output_valid
input output_ready
input input_finished
```


RIGHTSLINK

```
output output_finished
```

A ready-valid handshake interface is used for both input and output to support arbitrary delays in the processing unit (caused by `while` loops) and in the input source and output sink. The `input_finished` signal is asserted continuously starting on the cycle immediately following the last input token handshake. The `output_finished` signal must be asserted continuously by the processing unit starting on the cycle immediately following the last output token handshake.

Two of our three state elements – registers and vector registers – are already present in RTL and are easy to compile. Our RTL BRAM module has write address, write enable, write data, and read address inputs, and a read data output. It is implemented with a standard pattern that FPGA vendor tools will generally synthesize to technology BRAMs. We rely on the tools to create BRAMs of arbitrary size out of one or more technology BRAMs. `wire` references are replaced with their full expansions in a preprocessing pass over a Fleet program, so our compiler does not need to deal with them; we rely on the underlying RTL compiler to perform common subexpression elimination and logic minimization for us.

We first describe the compilation process assuming that there are no `while` loops or stalls in the input source or output sink (i.e. the `input_valid` and `output_ready` IO signals are always true), and then describe how we handle these complications. At a high level, the compiler generates a two-stage virtual cycle pipeline, with the first stage performing BRAM reads and the second stage performing BRAM and register writes. Figure 5 shows an overview of the pipeline.

The Fleet language contains only two fundamental statements: assignment to any state element, and `emit`. The conditional operators – `if`, `else`, and `else if` – merely provide conditions that gate the execution of assignments or `emits`. For each register `r`, the compiler gathers all assignments to it in the program, along with their conditions. If an assignment has no condition, its condition is set to true. If an assignment is nested within multiple conditional blocks, its condition is simply the conjunction of all of the containing conditions. If one of the nesting blocks is a `while` loop, the `while` condition is treated as an `if` condition, since a `while` loop is simply an `if` block that our control logic executes multiple times. The next value `r_n` for a register `r` is selected from among the assigned values and the current value of the register, assuming at most one assignment condition is true. Next values for the registers in our histogram example are shown in lines 17-18 of Figure 4. A similar procedure is used for vector register assignments, BRAM writes, and `emits`.

BRAM Reads. The main difference in the compilation process for BRAM reads is the expression generation for the conditions and addresses, which performs result forwarding by using the next values instead of the current values to replace register references (example on line 29).

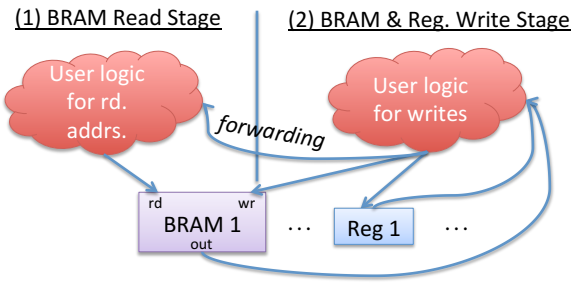

Figure 5. Two-stage virtual cycle pipeline

One issue is that the current virtual cycle may write the same address in a BRAM that the next virtual cycle reads. The read and write in this case are issued in the same real cycle, and the semantics of BRAMs for such conflicts are to return the old value for the read, which is not what we want. Our solution to this problem is to again use result forwarding, achieved with a register holding the last (address, data) pair written to each BRAM (definition on lines 10-11, use on line 31). If the user asserts, for a particular BRAM, that a virtual cycle never reads an address written by the previous virtual cycle (and this property can be checked by our software simulator on example input streams), then this extra register and logic can be elided.

`while` *Loops.* To handle `while` loops, we define a signal called `while_done` (line 15) that is set to true if there are no `while` loops in the program and to the negation of the disjunction of all loop conditions otherwise. The `input_ready` signal should only be true when `while_done` is true, as we only want to consume a new input token once the loop is complete and we are executing the post-loop virtual cycle. Finally, when generating the conditions for statements outside of `while` loops, we add the clause `while_done` to the conditions to prevent them from executing until the loop is complete (examples on lines 17-18 and 33).

Input and Output Stalls. To handle stalls in the input source or output sink, we make use of a register `v` (line 5) that indicates whether a virtual cycle is in progress. Register and BRAM writes cannot be committed unless we are ready to finish the current virtual cycle (lines 19-26). Stalls also require us to change the read input of BRAMs to depend on the current values of registers and vector registers instead of the next values (line 30) so that the BRAM read output does not change until the next virtual cycle. Lines 39-43 show the logic for accepting a new input token in the presence of stalls.

RIGHTSLINK

5 The Fleet Memory Controller

The previous section described the compilation of a single processing unit written in the Fleet language to RTL. The complete RTL that is synthesized to an FPGA bitstream for an Fleet application includes many identical copies of the processing unit as well as a soft memory controller that feeds and drains them. Our memory controller is designed differently from those of other FPGA programming frameworks, which are designed for the case of a single processing unit. These frameworks' memory controllers rely on their processing units having high-throughput input and output buffers composed of many individual BRAMs. We could take this approach, which would allow a simple strategy of performing a large sequential transfer for one processing unit at a time to saturate memory bandwidth. Unfortunately, it would almost certainly exhaust BRAM resources with even a moderate number of processing units on any real FPGA platform. To save BRAM resources, each of our processing units has only low-throughput input and output buffers, so our memory controller must feed multiple processing units in parallel to saturate memory bandwidth, requiring a new design.

Our current memory controller implementation targets an AXI4 memory interface with a 512-bit data bus. Our system could easily be ported to other memory interfaces with different bus widths, as long they support a ready-valid handshake protocol. We support an arbitrary number of AXI4 channels; the processing units are simply divided among the channels, and a separate input and output controller is instantiated for each of the channels. No further coordination is needed among the separate channels.

At a high level, the input and output controller for a channel operate in round robin fashion, checking with each processing unit in turn to see whether it is ready for new input or has output available. The controllers make DRAM requests at the granularity of multiples of the data bus width; larger multiples (a larger burst size) lead to better DRAM performance. Each processing unit has BRAM-based input and output buffers that have capacity equal to the burst size being used, so that communication with the input and output controllers can occur at the granularity of the burst size even though the core processing unit deals with individual tokens. Input and output stalls occur in the processing unit when the input buffer is empty or the output buffer is full. As noted above, the input and output buffers' data port widths must be small multiples of the native BRAM data port width (36 on the Amazon F1) to avoid excessive BRAM usage.

There are two key optimizations required to drive the memory system at its full rate.

Asynchronous Address Supply. First, due to high DRAM latencies, input and output addresses must be supplied well in advance of when their corresponding data packets are to be received or sent. Conceptually, this is easy for us to do because our processing units operate in streaming fashion, reading and writing DRAM locations only sequentially, so their upcoming input and output addresses are completely predictable. Both the input and output controller have a separate addressing unit that also operates in round robin fashion across the processing units but is several steps ahead of the data transfer unit, submitting addresses to the AXI4 interface. The input addressing unit skips over processing units that have already finished consuming input, and the output addressing unit skips processing units that have produced their last output.

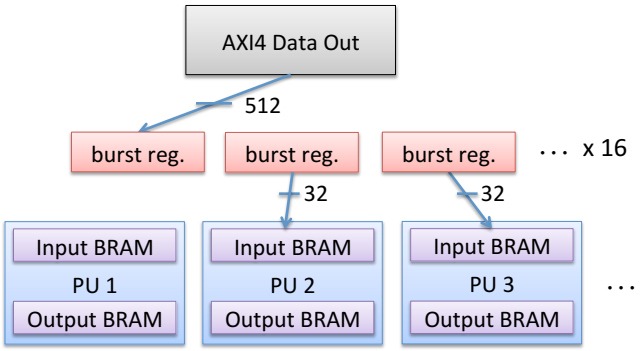

Figure 6. Burst registers for input controller

The AXI4 interface will return input data blocks in the same order that input addresses are supplied, and expects the output controller to supply data blocks in the same order that output addresses are supplied. We tell the interface that it can commit output blocks to memory in any order, which can increase performance. The input and output addressing units can be blocking or nonblocking; blocking units wait for each processing unit to supply its next input or output address, while nonblocking units skip processing units that are not ready for new input or output. Our default behavior is to make the input addressing unit blocking, since processing units generally process input at roughly the same rate, while the output addressing unit is nonblocking, since processing units that perform filters can produce output at dramatically different rates. A nonblocking addressing unit requires extra register storage to inform the data transfer unit which processing units were skipped; this register storage can be reduced by limiting how far ahead of the data transfer unit the addressing unit can get.

Burst Registers for Parallel Data Transfers. One issue is that the processing units' input and output BRAM buffers have data port widths much smaller than 512 bits, so we must feed and drain data from multiple processing units at once to keep up with the maximum AXI4 bandwidth of 512 bits of input and output per cycle. To do this, we define $r = 512/w$ registers of size equal to the AXI4 burst size for both the input and output controller, where w is the data port width of the input and output buffers. (w and r can be different for

RIGHTSLINK

the input and output controller, but we assume that they are the same in the following discussion.) Figure 6 illustrates the function of these burst registers.

The registers for the input controller store the last r received input bursts, and they are drained in parallel to the corresponding processing units that requested them. Similarly, the registers for the output controller store the next r output bursts to be sent to the AXI interface; they are filled in parallel by the corresponding processing units and sent to the AXI interface in order. There is a tradeoff between burst size and logic resources used for the input and output controllers: a larger burst size improves memory bandwidth but requires more burst register resources and therefore reduces the logic resources available for the processing units. In our experiments we use a burst size of 1024 bits (two 512-bit transfers), which allows us to reach near peak memory bandwidth on the Amazon F1 while leading to input and output controllers that together take up about a tenth of the logic resources available on the F1. $w = 32$ and $r = 16$ in our F1 designs.

6 Implementation

Our compiler for the Scala-embedded Fleet language generates code in the Chisel [8] Scala-embedded RTL language. Having all of the language components in Scala simplified the implementation. Scala's support for operator overloading and passing code blocks into functions made it easy to provide a clean Scala-embedded syntax for Fleet. Its `case classes` made it easy to define AST nodes for the language, and its support for pattern matching simplified AST passes, such as collecting all of the assignments to registers and BRAMs. We were able to write our Chisel code generator and software simulator in under 1000 lines of Scala code.

We took advantage of Chisel's peek-poke testing interface to build a full-system simulation infrastructure for our stream processing applications. We can simulate stream processing hardware with an arbitrary number of AXI4 channels, processing units, input and output controller burst registers, and other parameters. We wrote Scala code that simulates an AXI4 controller, accepting address requests and pushing and pulling data blocks from the simulated logic. Our testing infrastructure cross-checks the results of full-system RTL simulation against the Fleet language software simulator.

Our code is available at https://github.com/jjthomas/Fleet.

7 Evaluation

We compare the performance of Fleet applications (written in the Fleet language) with CPU and GPU implementations in terms of absolute performance and performance per watt. We also compare them to implementations using a commercial HLS system in terms of memory system performance, processing unit area and performance, and lines of code required.

7.1 Applications

We first describe our six Fleet applications.

JSON Parsing. Our JSON field extractor unit reads a list of fields to extract (e.g. `a.b`, `a.c`) at the start of its input stream and then emits the values of those fields encountered in the potentially nested JSON records in the remainder of the stream. The processing unit stores a transition table in a BRAM with states representing the character in the target field set that stream is currently on. For each state, there are one or more expected next characters and a pointer to the next states if those characters are encountered. Most of the logic in the processing unit is for the state machine that decides if a field match has been reached and handles JSON control characters like `{`, `:`, and `"`.

Integer Coding. Our integer encoding unit compresses blocks of consecutive 32-bit integers. The block size is four integers in our experiments, and a BRAM buffer is used to store the current block during processing. Our logic tries sixteen different fixed width encodings for the block in parallel, storing integers that fit within the fixed width in a main section and rest in an exception section that is coded with variable-byte encoding or the best possible fixed width encoding, whichever is cheaper. This scheme is inspired by OptPFD and other expensive coding techniques described in [17]. Fast integer-specific compression techniques can be used for integer columns in columnar databases and data to be sent over the network in distributed computing systems.

Decision Tree. Our gradient-boosted decision tree evaluator first loads the gradient-boosted decision tree nodes, located at the start of the stream, into a BRAM. The remainder of the stream has datapoints consisting of a runtime-configurable number of 32-bit integers. Each datapoint is loaded into a BRAM and the gradient-boosted decision tree is evaluated on it, with the result emitted to the output stream.

Smith-Waterman. Our Smith-Waterman fuzzy matching unit takes a target string as input at runtime and computes the edit distance matrix between the target and the remainder of the stream. The distance matrix has dimension $n \times m$, where n is the stream size in characters and m is the target string size. The only storage required by our unit is m registers for a single row of the matrix corresponding to the current character in the input stream, since matrix values only depend upon values in the same and previous row. The unit emits the current index in the stream whenever any cell in the row exceeds a runtime-provided score threshold. Software reading the output stream can go back to the input stream at the emitted locations and reconstruct the exact matches. $m = 16$ in our experiments. Fuzzy matching is

RIGHTSLINK

useful for DNA sequencing applications as well as search applications such as ElasticSearch.

Regex. Our regex matching unit takes a compile-time specification of a regex using character matches, concatenation, alternation (|), and repetition (*) and generates a circuit that matches the regex. The circuit is constructed according to the procedure described in [20], using primarily single-bit register state and binary operations to simulate an NFA for the regex. Whenever the unit detects a match, it emits the index of the current character in the stream. In our experiments we use an email regex taken from an online regex benchmark [4].

Bloom Filter. A Bloom filter is a bitfield that can be used to test membership in a set of items with no false negatives. Our last processing unit computes and emits a Bloom filter for each block of items. Each item in a block is hashed with several different hash functions, and a bit in the BRAM-based bitfield is set for each hash. Using an in-memory Bloom filter to quickly test whether a key exists can save disk IOs and improve throughput in key-value stores.

7.2 CPU/GPU Comparison

To demonstrate the efficacy of using Fleet to perform streaming computations on FPGAs, we compare the performance of our Fleet applications to hand-optimized CPU (C) and GPU (CUDA) versions, which use the same token-based processing model and algorithms as our Fleet versions. On the CPU, each core processes a single stream, and on the GPU, each thread processes a single stream. Our GPU implementations buffer input and output tokens in registers to avoid repeated reads and writes to memory locations, which are slow due to the lack of a fast automatic cache. For our CPU experiments we use a c4.8xlarge instance on the Amazon EC2 cloud, which has 36 Haswell hyperthreads. For our GPU experiments we use a p3.2xlarge instance, which has a V100 GPU. The GPU experiments use 225,280 threads partitioned into thread blocks of 256 threads each, which we found lead to the highest throughput.

It was difficult to take advantage of CPU vector units in our applications. Vectorization can be performed across multiple streams or within the processing of a single stream. The problem with vectorizing across streams is that different streams may diverge in their control flow, which prevents vectorization altogether, or perform accesses into stream-local memory, which cannot be vectorized using the AVX2 instructions available on the c4.8xlarge. Vectorizing within the processing of a single stream is difficult because the processing is inherently serial, with each token affecting the state for the next token. One case where single-stream vectorization is easier is if the processing of a single token is vectorizable, as is the case with the Bloom filter application, which performs 8 identical hash computations for each token. The Bloom filter is the only application we were able to vectorize on the CPU. Vectorization on the GPU is handled dynamically by the architecture – any stream processing threads running in the same warp will be vectorized if they are executing the same instruction.

We ran our Fleet experiments as follows. For each Fleet application, we filled the Amazon F1 FPGA (Xilinx UltraScale+ vu9p) with as many processing units (PUs) as would fit and used all four available DDR3 DRAM channels. Each processing unit consumed 1 MB of data; the time to transfer data from the host DRAM to FPGA DRAM was not measured. The logic clock was 125 MHz for all applications, which reduced power and allowed the applications to easily meet timing. Since the power measurements provided by Amazon's F1 tools included only package power, we assumed a constant DRAM power of 12.5 W, which was the highest DRAM power we observed for any of our applications on the CPU. We also assumed a constant DRAM power of 12.5 W for the GPU, since power measurements from the Nvidia tools included the sum of package and DRAM power but not the individual components.

The results of our comparisons are shown in Figure 7. Fleet outperforms the CPU in both absolute performance and performance per watt in all cases and outperforms the GPU in performance per watt in all but one case. Since the integer coding application has variable runtime in Fleet and on the CPU/GPU depending on how compressible its input is, experimental results for the application are the average of 5 runs with input integers drawn uniformly from the ranges [0, 2^5), [0, 2^{10}), [0, 2^{15}), [0, 2^{20}), and [0, 2^{25}). With the exception of integer coding, all of the applications produce substantially less output than input, with JSON parsing reducing its input by 80% and the rest reducing their input by more than 90%. JSON parsing, Smith-Waterman, regex, and Bloom filter were all bound on memory controller performance; if the memory controller and underlying DRAM system were arbitrarily fast these applications could have hit their theoretical computational throughputs of 64, 48, 88, and 40 GB/s, respectively.

We identify two primary reasons for our improvements: ability to handle divergence across streams and fusion of multiple CPU/GPU instructions into a single clock cycle's worth of work on the FPGA. As discussed above, stream divergence prevents the effective utilization of CPU and GPU vector units. Most of our workloads exhibit sufficient control flow and memory access divergence across streams to prevent vectorization with the AVX2 instruction set on CPUs, and two of our workloads – JSON parsing and integer coding – lose significant performance due to control flow divergence on the GPU. In particular, when running the JSON parsing experiment on the GPU with identical data for each stream, performance improves by 2.33×, and when doing the same for the integer coding experiment, performance improves by 1.25×. Similarly, when turning off AVX2 vectorization within the processing of each stream for the Bloom filter experiment

RIGHTSLINK

App	Fleet # PUs	Fleet Perf GB/s	Fleet Perf/W (w/ DRAM)	CPU Perf GB/s	CPU Perf/W (w/ DRAM)	GPU Perf GB/s	GPU Perf/W (w/ DRAM)	Fleet vs. CPU Perf/W (w/ DRAM)	Fleet vs. GPU Perf/W (w/ DRAM)
JSON Parsing	512	21.39	1.19 (0.70)	6.11	0.03 (0.03)	25.23	0.14 (0.13)	42.03× (26.24×)	8.57× (5.41×)
Integer Coding	192	10.99	0.73 (0.40)	2.11	0.01 (0.01)	31.04	0.16 (0.15)	78.19× (44.84×)	4.60× (2.67×)
Decision Tree	384	3.77	0.24 (0.13)	2.01	0.01 (0.01)	102.17	0.40 (0.38)	23.77× (14.06×)	0.59× (0.35×)
Smith-Waterman	384	24.62	1.37 (0.81)	0.68	0.003 (0.003)	29.41	0.15 (0.14)	444.67× (274.95×)	9.28× (5.82×)
Regex	704	27.24	1.51 (0.89)	3.25	0.02 (0.02)	73.59	0.36 (0.34)	95.54× (59.47×)	4.18× (2.62×)
Bloom Filter	320	24.21	1.15 (0.72)	12.03	0.05 (0.05)	13.50	0.12 (0.11)	22.43× (14.81×)	9.55× (6.66×)

Figure 7. Fleet on Amazon F1 vs. CPU/GPU

App	Fleet LoC	CUDA LoC
JSON Parsing	201	165
Integer Coding	315	155
Decision Tree	74	63
Smith-Waterman	55	45
Regex	35	65
Bloom Filter	100	58

Figure 8. Lines of code for Fleet and CUDA

Memory Controller Optimizations	Perf GB/s
None	0.98
Async. Addr. Supply	1.88
Async. Addr. Supply & Burst Regs.	27.24

Figure 9. Impact of memory controller optimizations

on the CPU, performance drops by 3.79×, suggesting that the other applications that cannot take advantage of AVX2 are losing significant potential performance.

The second reason for improvement is the fusion of multiple CPU/GPU instructions into a single FPGA cycle. This is particularly effective in cases where each virtual cycle has high computational intensity, as is the case in JSON parsing, integer coding, Bloom filter, Smith-Waterman, and regex with their numerous parallel comparisons and arithmetic operations for each input token. In contrast, while its computational intensity per input token is high, the decision tree application does only one comparison for each BRAM read, meaning that its computational intensity per virtual cycle is low and it is primarily bound on aggregate BRAM throughput. GPUs have substantially higher local memory and register throughput than the F1 FPGAs, explaining their excellent performance on this workload. While Fleet is still able to outperform the CPU on this workload, the speedup is smaller than for the other applications.

Fleet Developer Productivity. Finally, in Figure 8, we show the number of lines of Fleet code required for each application compared to the amount of CUDA code (which is similar to the amount of code for CPU and HLS). For Fleet, GPU, CPU, and HLS stream programming, the main challenge for the programmer is determining the logic for each stream; concerns like parallelization and DRAM access are handled by the underlying framework or architecture. The number of lines of code is similar between Fleet and CUDA, with the integer coding example requiring substantially more code because dynamic shifts are expensive in hardware, so output tokens were 8 bits instead of 32 bits and managing the division of output words into 8-bit chunks was fairly complex. We hope to add library code to Fleet to simplify this and other common patterns. The regex example takes fewer lines of code in Fleet because we count the lines of code in a Scala program that generates a circuit based on the input regex, while in CUDA the state machine for the email regex is fully elaborated.

7.3 Memory System Performance

To demonstrate the value of our memory controller optimizations (Section 5), we synthesized versions of the input controller without some of the optimizations and measured their performance. The application we used was a simple processing unit that drops all of the input tokens and produces no output. This application allowed us to isolate the performance of the input controller, since output controller performance should be symmetric. We first synthesized an input controller with synchronous address supply and only $r = 1$ burst registers. We then synthesized a controller with asychronous address supply but still $r = 1$ burst registers. Finally, we synthesized a full controller with asynchronous address supply and the ideal $r = 16$ burst registers. Figure 9 shows the results, demonstrating that asynchronous address supply provides a 1.9× performance benefit and burst registers provide an additional 14.5× performance benefit.

In terms of absolute performance, our input controller achieves 85% of the theoretical maximum bandwidth of 32 GB/s (one 512-bit transfer each 8 ns cycle for each of the four memory channels) and 91% of the 30.1 GB/s bandwidth we measured by reading from each of the memory channels

RIGHTSLINK

with the maximum burst size of 64 512-bit transfers. When adding our output controller and producing the same amount of output as input read, we achieve a performance of 11.38 GB/s, which is 69% of the peak measured bandwidth with input and output burst sizes of 64. The output controller's performance is lower because we have not yet implemented some optimizations relating to burst register utilization. The output controller was less critical for our experimental performance because most applications produced significantly less output than input.

7.4 HLS Comparison

In addition to comparing against the streaming performance of other hardware platforms, we consider other programming frameworks for FPGAs that could provide similar streaming performance to Fleet. The primary high-productivity FPGA programming framework deployed today is high-level synthesis (HLS), which takes OpenCL or C++ input along with pragma directives for the compiler and generates RTL. We consider a commercial HLS tool that takes OpenCL input and targets Xilinx devices, including the Amazon F1 FPGAs, and show that it has poor memory system and processing unit performance for multi-stream applications like ours.

Memory Controller Performance. We first consider the HLS system's memory controller generation abilities. We wrote a simple OpenCL application to compute the sum of the integers in 16 separate streams in a manner similar to Fleet. This application produces very little output, allowing us to isolate input controller performance, since output performance should be symmetric. We structured the OpenCL code as follows. Using a loop over the 16 streams, we loaded the next 1024-bit chunk of data (the same burst size we use for Fleet on the F1) for each stream into a local array (BRAM) for the stream. The local arrays had data port widths of 32 bits, which matches the port width we use for Fleet processing unit input buffers on the F1. We tried to both unroll and pipeline this loop, and used the special 512-bit `uint16` type for the global memory transfers as recommended by the tool documentation. We then had a second loop over the 16 streams where we computed the sum of the integers in the local array for each stream and added it to the running sum for the stream, which was stored in a register. This loop was unrolled.

Pipelining the first loop lead to a processing throughput of 524.84 MB/s, and unrolling it lead to a throughput of 675.06 MB/s. The experiment used only one out of the four available DRAM channels on the F1 platform. The pipelined throughput is thus 13.0× less than our memory controller's single-channel input throughput of 6.8 GB/s, and the unrolled throughput is 10.1× less. Essentially, the HLS tool was unable to figure out that it should fill multiple streams' local arrays simultaneously as our memory architecture does, instead filling them serially. We tried to use a larger burst size than 1024 bits to improve performance slightly, but even the largest burst size of 32,768 bits did not help, in fact reducing performance slightly. Even if there are optimizations we missed, there is a limit of 64 bits read from memory per cycle with the serial transfer approach, or a throughput of 1 GB/s (6.8× lower than ours) assuming a 125 MHz clock, since each processing unit's local array has only two 32-bit data ports. Using port widths larger than 32 bits is infeasible since the native BRAM port widths on the F1 platform are 36 bits, so larger ports would simply use more BRAM resources and limit the number of processing units that could fit on the chip. All six of our applications have total throughputs on the F1 well above this HLS system's maximum 4-channel F1 memory throughput of 2.7 GB/s, which is an upper bound on the performance of any HLS implementation of these applications.

Processing Unit Performance. The other major problem with the commercial HLS tool is that the OpenCL language is not an effective representation for token-based streaming programs. If we simply take the CUDA code we wrote for the JSON parsing and integer coding applications and adapt it slightly to be valid OpenCL, the HLS tool generates logic with initiation intervals (cycles per token) of 15 and 18, respectively. The Fleet versions of these applications take 1 cycle per token and 3-8 cycles per token, respectively. The primary reason is that the tool must make worst-case assumptions about the mutual exclusivity of program statements that may have BRAM resource conflicts. In the case of the JSON parsing and integer coding programs, there are many emits to the output buffer, but they are all mutually exclusive. Consider the following program snippet:

```
if (state == 0) {
  output_buf[output_idx++] = 0;
}
if (state == 1) {
  output_buf[output_idx++] = 1;
}
```

Assuming that the BRAM `output_buf` has only one write port, the HLS tool's OpenCL compiler infers a pipeline with an initiation interval of two for this code. If the second block was wrapped in an `else if` rather than an `if` (or if the compiler had a simple static analyzer) the initiation interval would be one instead, but it is often hard to write complex programs in a way that makes mutual exclusivity of BRAM reads or writes obvious. The Fleet language makes mutual exclusivity of BRAM reads and writes a requirement, sparing the compiler the need to perform complex static analysis and allowing it to reliably schedule program logic into one cycle. It is possible to add dependency pragmas to the OpenCL code to inform the compiler of mutual exclusivity, but these did not work when we tried them in the commercial tool, and the use of pragmas also demonstrates the mismatch of the OpenCL abstraction to the FPGA streaming problem. In

RIGHTSLINK

general, our streaming programs maintain fairly complex state machines and have many branches, and it appears that the commercial HLS tool is optimized for programs with more regular control flow and BRAM access.

Beyond poor initiation intervals, the commercial HLS tool also produces processing units with high resource usage, likely because of extra logic for more complex pipelines and conservative estimation of bitwdiths from OpenCL types such as `uint` and `uchar`. Excluding the logic for AXI4 communication, the naive OpenCL versions of JSON parsing and integer coding (without pragmas or custom-bitwidth types) consume about 4.6× and 2.8× more F1 logic cells than the Fleet versions, respectively.

8 Related Work

There are two major classes of work related to ours.

FPGA Programming Frameworks. There are several streaming-oriented or general-purpose frameworks that can simplify the development of streaming applications on FPGAs. In general, many of these frameworks and their associated languages have focused on high-throughput processing of a single stream, requiring programmers to write complex processing units that exploit pipeline and/or data parallelism and fill up the target FPGA [10, 19]. Due to their focus on single-stream applications, they do not feature the high-performance memory controllers required to support many serial stream processing units operating on separate streams.

C-like HLS languages have gained commercial adoption, and we compare to one HLS system in detail in Section 7.4. In addition to its poor memory controller performance for multi-stream applications, the HLS system is unable to properly optimize serial processing units written in the OpenCL language, even when compiler directives are provided. The compiler for the Fleet language performs less automatic scheduling of BRAM accesses and has more predictable behavior for the state machine-style processing units we consider.

Spatial [16] is a general-purpose framework that offers a lower-level language abstraction closer to the Fleet language, but is still designed for single-stream parallelism. Rigel [14] and Optimus [15] allow programmers to specify graphs of high-level operators for processing a single stream and then generate hardware from them. These frameworks still require programmers to think about whether they have expressed sufficient parallelism and whether they will fully utilize the target FPGA's resources. Of course, any of these frameworks' languages could be used to generate a serial Fleet processing unit, as long the emitted RTL has the IO interface we describe in Section 4.

Custom Streaming Architectures. Finally, there have been a few custom processor architectures defined for streaming. In general, our goal is to enable developers to achieve hardware acceleration on commodity FPGA platforms such as the Amazon F1, rather than proposing a new hardware architecture. UAP [12] and UDP [13] are architectures with many independent stream processors designed for many of the same applications that we consider, including parsing and regex matching. They have a fixed instruction set, which prevents the fusing of multiple operations into a single cycle that is possible on a reconfigurable architecture. The Imagine family of stream processors [7] is designed more for graphics and multimedia applications and has a VLIW architecture with a C-like stream programming model.

9 Conclusion

We have presented Fleet, an FPGA programming framework for applications processing many independent streams of data in parallel. We have demonstrated that with Fleet FPGAs can outperform CPUs and GPUs on multi-stream applications with low developer effort. Our extensions to Chisel for token-oriented stream processing and our custom memory system for multi-stream parallelism are the key factors in our results. We are able to fit hundreds of stream processing units for applications such as JSON parsing, integer compression, and Bloom filter construction on the Amazon F1 FPGA and saturate its memory bandwidth.

Acknowledgments

This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, Intel, Microsoft, NEC, SAP, Teradata, and VMware—as well as Toyota Research Institute, Keysight Technologies, Amazon Web Services, Cisco, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] [n.d.]. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[2] [n.d.]. Intel FPGAs Power Acceleration-as-a-Service for Alibaba Cloud. https://newsroom.intel.com/news/intel-fpgas-power-acceleration-as-a-service-alibaba-cloud/.

[3] [n.d.]. Internal Memory (RAM and ROM) User Guide. https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_ram.pdf.

[4] [n.d.]. Languages Regex Benchmark. https://github.com/mariomka/regex-benchmark.

[5] [n.d.]. Memory Usage Guide for iCE40 Devices. https://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/MO/MemoryUsageGuideforiCE40Devices.ashx?document_id=47775.

[6] [n.d.]. UltraScale Architecture Memory Resources. https://www.xilinx.com/support/documentation/user_guides/ug573-

RIGHTSLINK

ultrascale-memory-resources.pdf.

[7] Jung Ho Ahn, William J Dally, Brucek Khailany, Ujval J Kapasi, and Abhishek Das. 2004. Evaluating the imagine stream architecture. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. IEEE, 14–25.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 1212–1221.

[9] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[10] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[12] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 533–545.

[13] Yuanwei Fang, Chen Zou, Aaron J Elmore, and Andrew A Chien. 2017. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 55–68.

[14] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 85.

[15] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. 2008. Optimus: efficient realization of streaming applications on FPGAs. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 41–50.

[16] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 296–311.

[17] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[18] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *Proceedings of the VLDB Endowment* 11, 11 (2018).

[19] Franjo Plavec, Zvonko Vranesic, and Stephen Brown. 2008. Towards compilation of streaming programs into FPGA hardware. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*. IEEE, 67–72.

[20] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 227–238.

[21] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

RIGHTSLINK